PROGRAMMIER-SPRACHEN

SONDERHEFT 5/1986 ÖS 100,-/sfr. 14,- Lit. 12000/hfl. 18,-/dkr. 68,- DM 14,-

HAPPY-COMPUTER

Markt&Technik

DAS GROSSE HEIMCOMPUTER-MAGAZIN

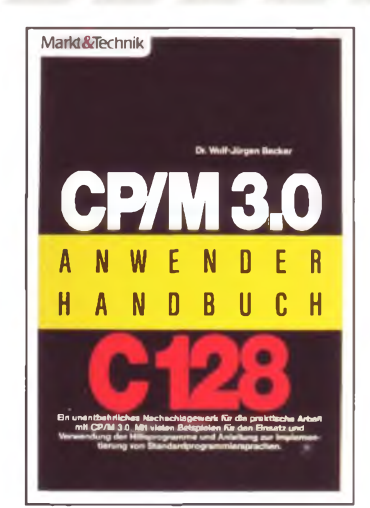


Bücher zum Commodore 128 PC

H. Ponnath
Grafik-Programmierung C 128
Februar 1986, 196 Seiten inkl. Disk

Die Programmierung von Grafik gehört zu den interessantesten Aufgaben, die man mit dem Commodore 128 PC lösen kann. Dieses Buch hilft Ihnen dabei! Das Themenfeld ist weit gespannt und behandelt unter anderem: hochauflösende und Mehrfarben-Grafik im C128-Modus. Alle BASIC 7.0-Befehle dazu werden detailliert besprochen und ihre Möglichkeiten und Grenzen gezeigt: die Programmierung von Sprites und Shapes; nützliche Assemblerprogramme (z.B. eine OLD- und eine MERGE-Funktion, die die modulare Programmierung unterstützt); die Videochips VIC und VDC und ihre Programmierung; eine Technik zur Erzeugung von selbstmodifizierenden Programmen.
Best.-Nr. MT 90202
ISBN 3-89090-202-2
DM 52,-/sFr. 47,80/öS 405,60

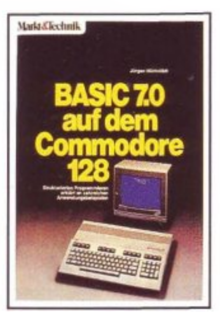


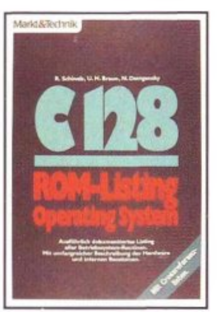


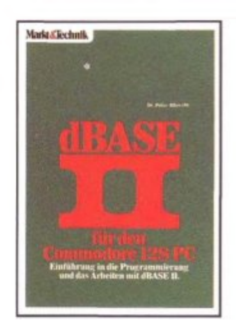
Prof. Dr. Wolf-Jürgen Becker
CP/M 3.0 Anwender-Handbuch C128
2. Quartal 1986, ca. 250 Seiten

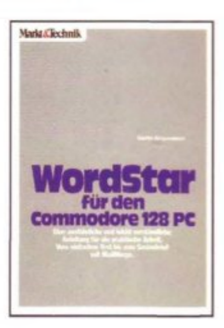
Wenn Sie Ihren Commodore 128 PC schon ganz gut im Griff haben und jetzt so richtig einsteigen wollen in die Möglichkeiten, die das leistungsstarke Betriebssystem CP/M 3.0 bietet, sollten Sie mal in dieses Buch schauen: es sagt Ihnen alles über den Aufbau einer Datenverarbeitungsanlage, Mikrocomputer, Programmiersprachen und Betriebssysteme im allgemeinen und über das Betriebssystem CP/M speziell auf dem C128 PC. Ausführliche Beschreibungen der CP/M-Befehle und ihrer Funktionen fehlen ebensowenig wie die umfassende Darstellung der Struktur von CP/M 3.0 auf dem C128. Im Kapitel über das Programmieren unter CP/M erfahren Sie dann, wie man das CP/M-Betriebssystem ändert, kommerzielle Software installiert und mit ihr arbeitet.
Best.-Nr. MT 90196
ISBN 3-89090-196-4
DM 52,-/sFr. 47,80/öS 405,60

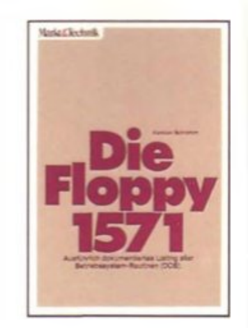












P. Rosenbeck
Das Commodore 128-Handbuch
1985, 383 Seiten

Dieses Buch sagt Ihnen alles, was Sie über Ihren C128 wissen müssen: die Hardware, die drei Betriebssystem-Modi und was die CP/M-Fähigkeit für Ihren Computer bedeutet. Aber Sie werden irgendwann Lust verspüren, tiefer in Ihren C128 einzusteigen. Auch dafür ist gesorgt: an einen Assemblerkurs, der Ihnen zugleich die Funktionsweise des eingebauten Monitors nahebringt, schließen sich Kapitel an, die mit Ihnen auf Entdeckungsreise ins Innere der Maschine gehen. Daß die Reise spannend wird, dafür sorgen die Beispiele, aus denen Sie viel über die Interna des Systems lernen können – bis hin zur Grafik-Programmierung.
Best.-Nr. MT 90195
ISBN 3-89090-195-6
DM 52,-/sFr. 47,80/öS 405,60

J. Hückstädt
BASIC 7.0 auf dem Commodore 128
1985, 239 Seiten

Das neue BASIC 7.0 des C128 eröffnet mit seinen ca. 150 Befehlen ganz neue Dimensionen der BASIC-Programmierung. Es ermöglicht dem Anfänger den einfachen und effektiven Zugriff auf die erstaunlichen Grafik- und Tonmöglichkeiten des C128; der Fortgeschrittene findet die nötigen Informationen für (auch systemnahe) Profi-Programmierung mit strukturierten Sprachmitteln.
An praxisnahen Beispielen (wie z.B. der Dateiverwaltung) zeigt der Autor auf, wie man die für den 128er typischen Merkmale und Eigenschaften (Sprites, Shapes, hochauflösende Grafik, Musikprogrammierung und Geräusche) optimal nutzt!
Best.-Nr. MT 90149
ISBN 3-89090-149-2
DM 52,-/sFr. 47,80/öS 405,60

R. Schineis, M. Braun, N. Demgensky
C128-ROM-Listing: Operating System
März 1986, 450 Seiten

Dieses Buch ist für alle Programmierer und Anwender gedacht, die mehr über ihren Commodore 128 PC wissen wollen: Eine Einführung in die Organisation und Wirkungsweise eines Mikrocomputers sowie eine detaillierte Beschreibung der Mikroprozessorfamilie 65XX bzw. 8502. Aufbau und spezielle Hardwareeigenschaften des C128 mit Beispielprogrammen. Ein umfangreiches, vollständig kommentiertes Assemblerlisting mit Cross-Referenzliste (Verweistabelle) umfaßt das komplette Betriebssystem mit dem 40/80-Zeichen-Editor sowie allen Kernel-Routinen.
Best.-Nr. MT 90221
ISBN 3-89090-221-9
DM 49,-/sFr. 45,10/öS 382,20

Dr. P. Albrecht
dBASE II für den Commodore 128 PC
1985, 280 Seiten

Das vorliegende Buch gibt nach einer kurzen Einführung in den Komplex »Datenbanken« eine Anleitung für den praktischen Umgang mit dBASE II. Schon nach Beherrschung weniger Befehle ist der Anwender in der Lage, Dateien zu erstellen, mit Informationen zu laden und auszuwerten.
Best.-Nr. MT 838
ISBN 3-89090-189-1
DM 49,-/sFr. 45,10/öS 382,20

Dr. P. Albrecht
Multiplan für den Commodore 128 PC
September 1985, 226 Seiten
Best.-Nr. MT 836
ISBN 3-89090-187-5
DM 49,-/sFr. 45,10/öS 382,20

G. Jürgensmeier
WordStar 3.0 mit MailMerge für den Commodore 128 PC
1985, 435 Seiten

WordStar ist ein umfangreiches und leistungsfähiges Textverarbeitungsprogramm. Doch bedeutet dies nicht unbedingt, daß es auch einfach zu bedienen ist. Hier setzt dieses Buch an: Es macht in vorbildlicher Weise mit allen Möglichkeiten von WordStar und MailMerge vertraut und ist damit eine ideale Ergänzung zum Handbuch.
Best.-Nr. MT 780
ISBN 3-89090-181-6
DM 49,-/sFr. 45,10/öS 382,20

K. Schramm
Die Floppy 1570/1571
2. Quartal 1986, ca. 400 S.

In der Floppy 1571 wurde ein völlig neues Floppy-Konzept verwirklicht: diese Floppystation ist in der Lage, mehrere verschiedene Diskettenformate zu verarbeiten. Dieses Buch soll es sowohl dem Einsteiger als auch dem fortgeschrittenen Programmierer ermöglichen, die vielfältigen Möglichkeiten dieses neuen Gerätes voll auszuschöpfen. Sämtliche Betriebsarten und Diskettenformate werden ausführlich erläutert.
Best.-Nr. MT 90185
ISBN 3-89090-185-9
DM 52,-/sFr. 47,80/öS 405,60



Bestellungen im Ausland bitte an den Buchhandel oder an untenstehende Adressen.
Schweiz: Markt & Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, ☎ 042/41 56 56
Österreich: Ueberreuter Media Handels- und Verlagsges. mbH, Alser Straße 24, 1091 Wien, ☎ 02 22/48 15 38-0

Irrtümer und Änderungen vorbehalten.



Unternehmensbereich Buchverlag
Hans-Pinsel-Straße 2, 8013 Haar bei München





Lieben Sie Pascal?

Das ist beileibe keine indiskrete Frage nach Ihrem Privatleben. Pascal ist eine Programmiersprache! Oder halten Sie's lieber mit Basic? Jeder, der sich intensiv mit seinem Computer beschäftigt, lernt schnell die Grenzen des eingebauten Basic kennen. Leicht zu verstehen und einfach zu programmieren – und eine unübersehbare Zahl von Listings zum Abtippen in fast allen Computermagazinen, die es am Markt gibt, machen Basic zu einer Bastion auf dem Heimcomputermarkt. Aber das muß nicht sein. Andere Programmiersprachen holen noch viel mehr aus Ihrem Computer heraus. Bloß, wie soll man sich in dem Babylon der Computersprachen zurechtfinden? C, B, Pascal, Forth, Ada..., die Namen der verschiedenen Sprachen und Dialekte sind Legion.

Welche Alternative muß man ergreifen, wenn man der eingebauten Computersprache überdrüssig ist? Um diese Frage zu beantworten, haben wir für Sie dieses Sonderheft zusammengestellt.

Am beliebtesten – oder am meisten in der Schule und Universität verlangt – sind zur Zeit Pascal, C und Forth. Zu diesen drei Sprachen finden Sie deshalb jeweils einen großen Einsteiger-Kurs. Lernen, das beschränkt sich aber nicht nur auf die Theorie. Deshalb sind alle Artikel mit vielen Beispiellistings zum Abtippen versehen. Denn nur, wer selbst ausprobiert, lernt die neue Sprache verstehen und effektiv einzusetzen. Vom Spiel bis zu Anwendungen, die das Leben eines Programmierers erleichtern, finden Sie alles, was das Herz begehrt.

Einen Forth- und einen Pilot-Interpreter zum Abtippen haben wir auch für Sie aufgetrieben. Denn nicht jeder will gleich teures Geld für eine Sprache ausgeben, mit der er sich dann vielleicht doch nicht anzufreunden vermag. Also erst mal reinschnuppern und anfangen in einer neuen Sprache zu programmieren.

Damit Sie wissen, was es alles an Sprachen für Ihren Computer zu kaufen gibt, finden Sie auf über fünf Seiten alle Sprachen, die für Heimcomputer im Handel sind. Viele davon haben wir getestet und wir sagen Ihnen, mit welcher Sie Ihren Commodore 64, Schneider CPC, Atari ST oder irgendeinen anderen Computer am besten füttern.

Anspruchsvolle Exoten – wie beispielsweise Prolog und Lisp – dürfen natürlich nicht fehlen in einem Heft, das sich »Programmiersprachen« auf die Fahne geschrieben hat. Über künstliche Intelligenz, was immer das heißen mag, erfahren Sie also auch einiges.

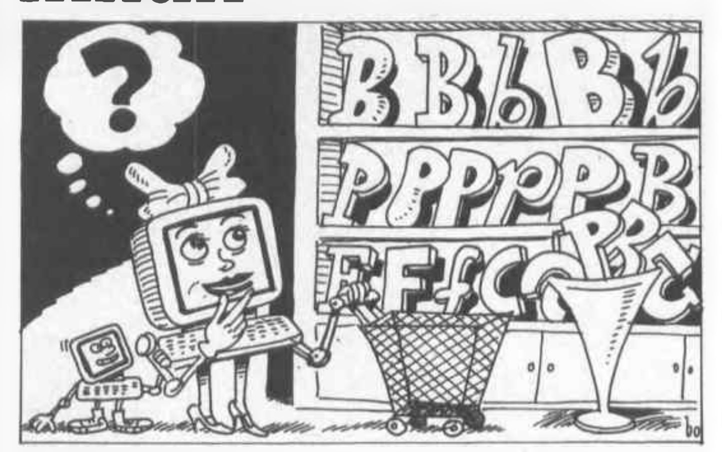
Programmiersprachen sind ein weites Thema, das mehr als nur ein Heft füllen würde. Deshalb sind wir auf Ihr Interesse gespannt. Schreiben Sie uns, ob Sie mehr über »Fremdsprachen« für Ihren Computers lesen wollen. Denn wie jede Ausgabe von Happy-Computer orientieren sich auch unsere Sonderhefte am Interesse unserer Leser. Wir wollen schließlich über das berichten, was Sie lesen wollen – Sie, unser Leser.

In diesem Zusammenhang möchte ich allen danken, die mir bei der Zusammenstellung dieser Ausgabe geholfen haben. Besonders den Lesern unter Ihnen, die uns ihr Wissen zur Verfügung gestellt haben. Machen Sie bitte weiter so mit.

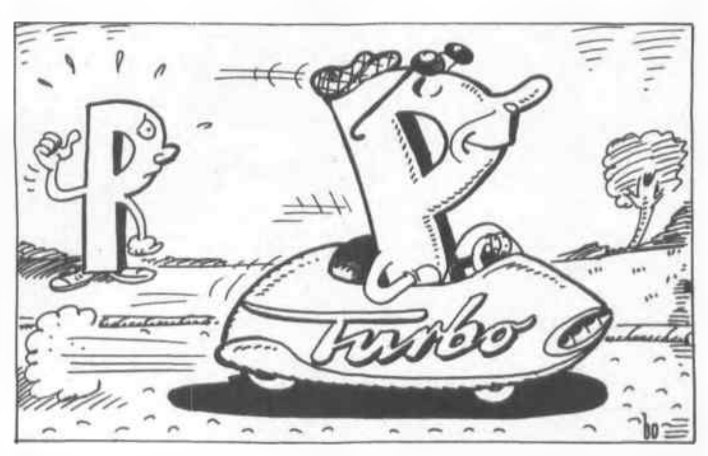
Zu diesem Sonderheft haben wir für einige Computer jeweils eine »Programmiersprachen-Schnupper-Diskette« zusammengestellt. Auf ihnen finden Sie Programme, die die verschiedenen Fähigkeiten der Sprachen offenbaren. Viel Spaß damit! Bei den Listings hingegen müssen Sie darauf achten, daß nicht jeder Compiler alles kann. Aber ausprobieren hilft immer – und schult das Verständnis für das Programmieren.

Ihr Andreas Hagedorn

Inhalt



Vor Jahren gab es nur eine Handvoll Programmiersprachen. Inzwischen steht man ratlos vor einer Vielzahl von Sprachen und ihren Dialekten. Unsere Marktübersicht gibt eine Hilfestellung bei der Auswahl. 156



Ein Spitzenreiter in jeder Beziehung: Turbo-Pascal. Ein enorm niedriger Preis, die Vorzüge hoher Geschwindigkeit und dennoch hohen Komforts machen diese Pascal-Version unschlagbar. 19



Wer kennt sie nicht: Pascal, die Sprache, die strukturierte Programmierung populär machte. Was alles dazugehört, wie Prozeduren, Funktionen, Datentypen etc., kommt hier zur Sprache. 26

Programmiersprachen

Pascal

C

Forth

Pascal-Kurs

Pascal-Listing

C-Kurs

C-Listing

Forth-Kurs

Forth-Listing

Pilot

Pilot-Listing

Modula

Künstliche Intelligenz

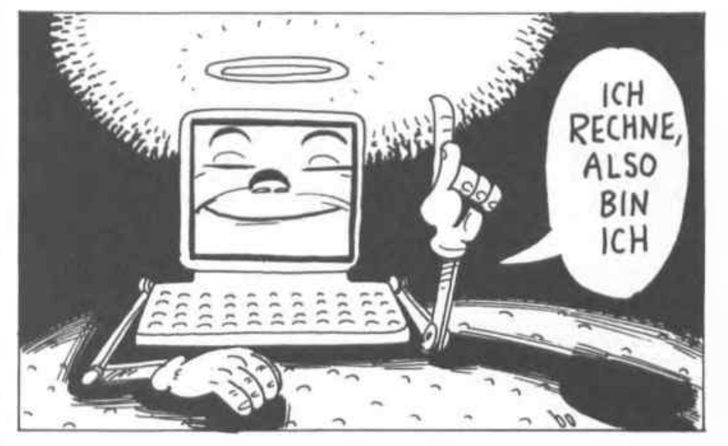
Marktübersicht



C ist eine faszinierende Sprache, die immer mehr ins Gespräch kommt. Mit Recht, denn sie bietet sowohl die vorbildliche Strukturierung von Pascal, als auch hohe Geschwindigkeit durch optimalen Aufbau ihres Codes. **78**



Forth, die Sprache, die ursprünglich von der NASA entwickelt wurde, setzt sich zunehmend auch auf dem privaten Sektor durch. Leistungsfähig, jedoch mit einem ungewöhnlichen Konzept, ist sie recht einfach zu erlernen. **110**



Hier geht es um Prolog, Lisp und Logo, die sogenannten Programmiersprachen für Künstliche Intelligenz und die »5. Computer-Generation«. Diese Sprachsysteme unterstützen die Fähigkeit, von Programmen dazuzulernen. **149**

Programmiersprachen – Babel läßt grüßen

Wer den Computer zu seinem Hobby oder Beruf macht, steht in mehrerer Hinsicht vor einer schwierigen Wahl. Er muß sich zwischen einer Vielzahl von Peripherie, Betriebssystemen und Programmiersprachen entscheiden. Hier zeigen wir Ihnen die wichtigsten Programmiersprachen, ihre Entwicklungsgeschichte und was sie können.

Kommunikation braucht Sprache. Wenn sich zwei Menschen miteinander unterhalten, benutzen sie dazu das gesprochene Wort oder, sofern sie nicht die gleiche Sprache verstehen, Hände und Füße. Dieses Prinzip läßt sich nicht ohne weiteres auf die Verständigung zwischen Mensch und Computer anwenden. Kommunikation ist der Datenaustausch zwischen mehreren Parteien, die einander verstehen müssen, jedoch ist ein Computer im Urzustand alles andere als verständig. Er besitzt lediglich eine mehr oder weniger große Anzahl von Fähigkeiten, die er sehr schnell ausführen, aber nicht selbständig koordinieren kann.

Befehl und Gehorsam

Programmiersprachen sind, im Gegensatz zur weitverbreiteten Meinung, kein Mittel, um sich mit dem Computer zu unterhalten. Sie dienen lediglich dazu, dem Computer über eine Kette von Befehlen mitzuteilen, was er Schritt für Schritt zu tun hat. Programmierung wurde lange Zeit unter dem Hauptaugenmerk der Mensch-Maschine-Kommunikation betrachtet. Der wichtigste Aspekt lag darin, zu Problemlösungen unter möglichst effizienter Nutzung der Maschinenkapazität zu gelangen. Seit Computer in jüngster Zeit eine stärkere Verbreitung erfahren haben, rückte jedoch ein zweiter Gesichtspunkt immer mehr in den Vordergrund: Die Programme werden komplexer und der Wartungsaufwand (Korrektur oder Erweiterung eines Programms) immer größer. Dadurch, daß an vielen Programmen große Teams über lange Zeiträume hinweg arbeiten, geraten Programmiersprachen auch mehr und mehr zum Kommunikationsmittel zwischen diesen Gruppen von Menschen. Das bedeutet, daß ein guter Programmierer immer seine Programme auch für seine Nachwelt verständlich gestalten, und seine Kunstfertigkeit nicht mit der Anwendung von Programmiertricks unter Beweis stellen sollte. Einem potentiellen Benutzer, der das Programm lesen und eventuell ändern muß, sollte die Funktionsweise möglichst schon beim Lesen klar werden.

Statt »Programmiersprache« wäre die Bezeichnung »Kommandosequenz« eigentlich richtiger. Was dem Computer befohlen wird, führt er geduldig und beliebig oft aus, einzige Bedingung ist ein fehlerfreies Programm. Einige Psychologen behaupten denn auch, die Beliebtheit von Computern sei darauf zurückzuführen, daß viele Menschen ihre diktatorischen Triebe beim Programmieren ausleben!

Wenden wir uns der Frage zu, die wohl jeden Computeranwender irgendwann einmal bewegt, nämlich, was denn welche Programmiersprache wohl zu leisten vermag.

Im Laufe der Computergeschichte, die seit den ersten Relaisrechnern Konrad Zuses nicht mehr als ein halbes Jahrhundert zählt, wurde eine Unzahl Programmiersprachen, Spracherweiterungen und Dialekte entwickelt. Dabei standen immer zwei Überlegungen im Vordergrund. Zum einen mußten die Hardwarevoraussetzungen berücksichtigt werden. Zum anderen orientieren sich die Programmentwickler an den zu bearbeitenden Problemstellungen und den Bedürfnissen der Anwender. Je nach Computertyp und Problemstellung werden vom Programmierer verschiedene Programmstrukturen (Module, Blockkonzept, Verbundtypen), spezialisierte Befehle und unterschiedliche Datentypen (Integer, Real, Complex, String, etc.) benötigt. Auf dieser Grundlage entwickelten sich neben den bekannten Sprachen eine unüberschaubare Anzahl Exoten, die meistens nur in einzelnen Universitäten eingesetzt wurden.

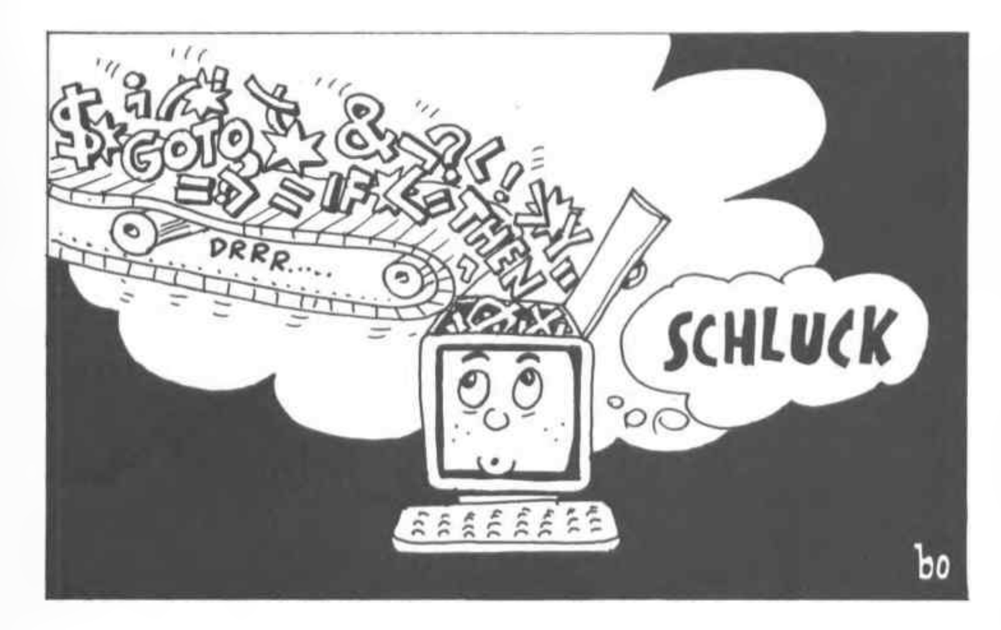
Vielfalt – Freud oder Leid?

An kaum einem Punkt scheiden sich die Geister in der Computerszene so stark wie in der Auswahl der Beurteilung der Programmiersprachen. Wer den Heimcomputer zu seinen Hobbies zählt, lernt in aller Regel zunächst fleißig Basic. Schließlich gehört ja der Basic-Interpreter zum Lieferumfang. Im Laufe der Zeit werden die eigenen Programme immer länger und unübersichtlicher, die einzelnen Programmteile sind durch ein unentwirrbares Geflecht von GOTO-Anweisungen miteinander verknotet (böse Zungen sprechen deshalb von »Spaghetti-Code«).

Mit dem wachsenden Bedürfnis nach Strukturierung, die in Basic nur jemand mit viel Selbstdisziplin erreicht, und nach Geschwindigkeit, die für Basic ein Fremdwort ist, sieht man sich nach Auswegen um. Dabei fühlt sich die eine Gruppe wie magisch von dem Begriff Assembler angezogen und begibt sich auf die unterste Sprachebene, um fortan in mühseliger Kleinarbeit Byte für Byte zu programmieren. Hiermit wird zwar ein Höchstmaß an Geschwindigkeit möglich, aber die Übersichtlichkeit kommt nach wie vor zu kurz. Die zweite Gruppe der Programmier-Gemeinde wendet sich deshalb modernen Hochsprachen zu, wie Pascal, Forth, Modula, Comal, Ada, C und so weiter. Diese Sprachen zwingen den Programmierer dazu, seine Programme modular (dies bedeutet, einzelne Aufgaben werden in »Paketen« oder »Modulen« zusammengefaßt) zu gestalten. Durch die so erreichte Übersichtlichkeit lassen sich Programme später von jedermann, Sprachkenntnisse vorausgesetzt, nachvollziehen und ändern. Zudem

sind einige dieser Sprachen sehr assemblernah, wie zum Beispiel Forth, womit auch Geschwindigkeit kein Problem mehr ist. Moderne Programmiersprachen unterstützen also Programmstrukturen und gehen ebenso mit Daten- und Kontrollstrukturen problemgerecht um.

Dennoch werden im professionellen Bereich (Universitäten, Verwaltung) »klassische« Sprachen bevorzugt, wie PL/1, Cobol und Fortran. Es sprechen auch gute Gründe dafür: So sind die neueren Programmiersprachen oftmals auf der vorhandenen Hardware noch nicht verfügbar, oder sie vertragen sich mit bereits vorhandenen Softwarekomponenten nicht. Ein anderer Faktor sind die Vorkenntnisse des Wartungspersonals und und und .... Jeder Informatikstudent, jeder Praktiker kann diese Reihe beliebig fortsetzen. Und schließlich ist gute Software immer noch der Triumph des Programmierers, nicht der der Programmiersprache.



Seit das Betriebssystem CP/M auf Computern wie dem Commodore 128, Schneider CPC, oder dem Atari ST einen neuen Frühling erlebt, werden diese Klassiker neuerdings auch auf Heimcomputern interessant. Sehr wahrscheinlich wird sich ein breiter Anwenderkreis hierfür finden. Welcher Programmierer begrüßt es nicht, wenn er seine Produkte teilweise in der »guten Stube« austesten kann?

Eine Sonderstellung unter den Programmiersprachen nehmen die Sprachen für »künstliche Intelligenz« (KI) ein. Deren bekannteste Vertreter heißen Lisp und Prolog. KI-Sprachen bauen auf einem grundlegend neuen Konzept auf, bei dem der Programmierer dem Computer im Dialog sein Problem mitteilt. Wegen des hohen Speicherbedarfs sind KI-Programme auf Microcomputern nur sehr eingeschränkt einsatzfähig. Dies wird sich jedoch bald ändern. Man darf gespannt sein, wie sich KI auf den 16-Bit-Computern entwickeln wird, die ja nicht nur in punkto Schnelligkeit, sondern auch im Speicherangebot einen ganz neuen Standard setzen. Der Künstlichen Intelligenz ist in diesem Sonderheft ein eigener Beitrag gewidmet.

Fassen wir zusammen: Sinnvoll lassen sich die Programmiersprachen in vier Gruppen unterteilen:

1. Assemblersprachen: Sie bieten den Vorteil, daß sie die Möglichkeiten der Hardware optimal ausnutzen. Assembler versteht jeder Prozessor unmittelbar. Jede höhere Programmiersprache muß beim Programmablauf erst in Assemblercode übersetzt werden und ist deshalb weniger universell. Gegen Assembler sprechen die mühselige Programmierung, die Gebundenheit an Prozessor und Hardware sowie die schwierige Wartung.

2. Klassische Hochsprachen: Sie sind zur Zeit am weitesten verbreitet, aber genügen wegen ihrer altertümlichen Konzeption den Anforderungen moderner Programmierung nicht mehr.

3. Moderne Hochsprachen: Sie können sich gegen die »Alteingesessenen« nur langsam durchsetzen. Sie bieten Strukturierungshilfen, ausgeprägte Möglichkeiten der Datenbeschreibung, und unterstützen die Selbstdokumentation des Programmtextes durch eine angemessene Verbalisierung.

4. KI-Sprachen sind heutzutage noch Gegenstand intensiver Forschungen und stellen eine völlig neue und andersartige Klasse von Programmiersprachen dar. In diesem Beitrag wollen wir uns deshalb nicht weiter mit ihnen beschäftigen.

Wie Sie gesehen haben, ist der Sprachenwald immer noch sehr dicht, auch wenn man sich auf die Auswahl der wichtigsten Programmiersprachen beschränkt. Ein bedeutendes Auswahlkriterium sind die qualitativen Merkmale. Bevor wir diese besprechen, noch zu einigen zentralen Begriffen:

Algorithmus – Jedem Programm liegen ein oder mehrere Lösungsverfahren zugrunde, oft auch als Algorithmus bezeichnet. Ein solcher Algorithmus ist definiert als eindeutige und vollständige Vorschrift zur Lösung einer Problemklasse mit einer Abfolge von Schritten, die in einem endlichen Zeitraum ausgeführt werden.

Daten heißen dabei die Objekte, die der Algorithmus bearbeitet.

Programm nennt man folgerichtig die maschinengerechte Aufbereitung der Daten und des Algorithmus. Diese Aufbereitung geschieht mit Hilfe einer künstlichen Sprache, eben der **Programmiersprache**. Diese setzt sich aus einer Menge von Zeichen zusammen, die ihrerseits nach bestimmten Regeln zusammengesetzt werden können. Die somit geschaffenen Sprachelemente werden von der Maschine unmittelbar (als Maschinensprache) oder unter Zuhilfenahme von Übersetzungsprogrammen (als Hochsprache) »verstanden«. Der Übersetzer stellt nichts weiter dar, als einen speziellen Algorithmus, der in der Lage ist, alle Befehle einer Hochsprache in den entsprechenden Assemblercode zu transformieren. Er ist grob vergleichbar mit einer Bibliothek von kleinen Assembler-Unterprogrammen, wobei jedem Befehl der Hochsprache eines dieser Unterprogramme zugeordnet ist.

Compiler und Interpreter

Übersetzungsprogramme gliedern sich in zwei Typen, die die gleiche Aufgabe auf unterschiedliche Weise erfüllen. Als erstes sind die **Compiler** zu nennen. Sie tauchten auch in der geschichtlichen Entwicklung zuerst auf. Der Compiler übersetzt den Programmtext (Quellcode) in einem oder mehreren Durchgängen (Passes) komplett in Assemblercode (Objektcode). Der Compiler selbst wird daher beim Programmlauf nicht mehr benötigt. Natürlich benutzt man für unterschiedliche Computer, mit verschiedener Hardware und Maschinensprache auch unterschiedliche Compiler.

Der zweite im Bund der Dolmetscher nennt sich **Interpreter**. Er ist der fleißigere von beiden. Zunächst muß der Quellcode direkt im Arbeitsspeicher des Computers abgelegt werden. Sodann beginnt das Interpretieren. Das heißt nichts weniger, als daß der Interpreter während des Programmlaufs Befehl für Befehl holen muß. Natürlich ist diese Vorgehensweise in höchstem Maße unökonomisch und langsam. Während ein Compiler die ersten drei der genannten vier Arbeiten nur genau einmal ausführen muß, beschäftigen sie den Interpreter bei jeder Programmwiederholung aufs neue.

Zu Beginn des Computerzeitalters, als Rechenzeit noch sehr teuer war, wurden daher ausschließlich Compiler entwickelt. Interpreter konnten sich erst mit höheren Prozessorleistungen durchsetzen. Sie werden auch heute noch in Profikreisen wegen ihres gemächlichen Arbeitstempos verschmäht.

Die Qualität von Programmiersprachen

Um beurteilen zu können, welche Eigenschaften gute Programmiersprachen charakterisieren, wenden wir uns zunächst der Frage zu, welche Anforderungen an ein hochwertiges Programm zu stellen sind.

Die Forderung nach Fehlerfreiheit erscheint auf den ersten Blick banal. Die Erfahrung zeigt nämlich, daß 100%ig korrekte Programmsysteme ab einem gewissen Umfang kaum noch möglich sind. Hat ein Programm eine gewisse Komplexität erreicht, ist es fast unmöglich, seine Korrektheit zu beweisen, also Testdurchläufe zu finden, die tatsächlich alle Eventualitäten berücksichtigen. Als Maß für die Korrektheit eines Programmes wird deshalb häufig von der Zuverlässigkeit gesprochen. Diese gibt die Wahrscheinlichkeit an, mit der ein Programm für eine Zahl von Anwendungsfällen in einer bestimmten Zeitspanne fehlerfrei arbeitet. Gerade die jüngste Vergangenheit hat hierfür im Bereich der Mikrocomputer einige negative Beispiele geliefert. So konnten Fehler in einigen Betriebssystemen oftmals erst nach der Auslieferung der neuen Geräte beseitigt werden.

Verständlichkeit deckt sich mit Forderungen nach Lesbarkeit, Überschaubarkeit und Selbstdokumentation. Programme sollten sich, sobald sie eine gewisse Länge überschreiten, in funktionelle Einheiten gliedern. Andernfalls wird der Programmierer oft, noch während er an ein und demselben Programm arbeitet, an der selbst produzierten Unordnung scheitern. Man differenziert unter dem Aspekt der Verständlichkeit zwischen der statischen und der dynamischen Programmstruktur. Die statische Strukturgestaltung dient dem Ziel, ein übersichtliches Layout des Programmtextes zu gestalten. Hiermit wird der menschlichen Wahrnehmung Rechnung getragen, die sehr stark auf optischen Wahrnehmungen beruht. Ergänzend trägt die dynamische Strukturierung dazu bei, daß Programmabläufe unmittelbar aus dem Quelltext ersichtlich werden.

Ein Faktor der an diese Zusammenhänge anknüpft, ist die Änderbarkeit von Programmen. Wartungsfreundlichkeit bedeutet einerseits, daß Programme leicht an modifizierte Aufgabenstellungen angepaßt werden können. Andererseits spielt die Portabilität eine Rolle, wenn ein Programm zum Beispiel in einer neuen Softwareumgebung lauffähig gemacht werden soll.

Universalität sollte es einem guten Programm ebenfalls ermöglichen, ähnliche Aufgabenstellungen und Abwandlungen zu lösen.

Im Zusammenhang mit der Benutzerfreundlichkeit sollten Programme einen Dialog mit dem Benutzer ermöglichen und Eingabefehler abfangen, ohne falsche Ergebnisse oder gar Systemabstürze zu liefern.

Effizienz hat im Laufe der Entwicklung stark an Bedeutung verloren. Zu Zeiten, da Speicherplatz noch Mangelware war, galten die kürzesten Programme als die besten. Das ging natürlich zu Lasten der Übersichtlichkeit. Effizienzstreben wird heute daher nur noch mit Skepsis betrachtet.

Die hier genannten Qualitätsmerkmale stehen offensichtlich in positiver und negativer Wechselwirkung zueinander. So geht Übersichtlichkeit meistens zu Lasten der Effizienz, verbessert dagegen aber die Änderbarkeit.

Beginnen wir jetzt damit, die Anforderungen an eine ideale Programmiersprache zu formulieren. Die Qualität einer Computersprache läßt sich danach beurteilen, inwieweit sie die Entwicklung von Programmen unterstützt, die den uns bekannten Anforderungen entsprechen. Die folgenden skizzierten Merkmale müssen im engen Zusammenhang miteinander betrachtet werden.

Beginnen wir mit einem Punkt, der besonders den Einsteiger interessieren wird:

Die Erlernbarkeit einer Sprache hängt wesentlich von deren Struktur und Umfang ab. Sie ist sehr viel einprägsamer, wenn die Zahl der Schlüsselwörter gering und das Sprachkonzept durchgängig ist. Ebenso gewährleistet eine einfache Benutzung, wenn der Programmierer sein Problem bequem mit einer breiten Palette von Ausdrucksmöglichkeiten lösen kann.

Die Einheitlichkeit ist ein ebenso wichtiger wie unscharfer Begriff. Er soll im großen und ganzen bedeuten, daß für eine bestimmte Leistung möglichst nur genau ein Sprachmittel zur Verfügung steht.

Kriterien für eine ideale Sprache

Kompaktheit steht mit Einfachheit in Einklang. Hierbei ist nicht von der »Würze der Kürze« die Rede, sondern vielmehr von der Mächtigkeit der Sprachkonzepte. Der Bauplan der Sprache muß eine geringe Anzahl verschiedener Grundkonzeptionen aufweisen, wie mathematische Operationen, Ein- und Ausgabe, Datenstrukturierung. Andererseits soll Kompaktheit ein gewisses Maß an Redundanz (= alles was man in der gleichen Sprache auf andere Weise auch darstellen kann) an der richtigen Stelle nicht verhindern. Dies fördert die Verständlichkeit und Zuverlässigkeit. Beispielsweise sind Datentypen im Grunde genommen redundant, doch wer möchte sich schon mit einer Sprache herumschlagen, die ausschließlich den Datentyp »Zeichen« kennt?

Die Sprache Basic wird von vielen als katastrophal eingestuft. Das hat einen guten Grund: Die Forderung nach Lokalität wird von Basic so gut wie gar nicht unterstützt. Mit diesem Begriff ist gemeint, daß die Teile einer bearbeiteten Aufgabe, die logisch zusammengehören, auch im Programmtext in physischer Nachbarschaft stehen sollten. Die berühmt-berüchtigten Sprungbefehle bewirken jedoch das genaue Gegenteil.

Ein ganz anderes Kriterium ist die Sicherheit der Programmiersprache. Dazu zählt das Unterstützen der Fehlerfreiheit eines Programmes durch Sprachelemente. »On Error Goto« ist ein solcher weitverbreiteter Befehl. Des weiteren sind Sprachelemente zu nennen, die die Testphase des Programms fördern.

Der »Wildwuchs« der Implementierungen (= Anpassungen eines bestimmten Computers) bei Programmiersprachen, insbesondere bei den älteren, ist hinlänglich bekannt. Dialekte, Erweiterungen, aber auch Einschränkungen beeinträchtigen die Portabilität von Programmen im besonderen Maße. Mit der Standardisierung

Fortsetzung auf Seite 10

PROGRAMM-SERVICE



Bestellungen in der Schweiz: Markt&Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Tel. 042/41 56 56
Bestellungen in Österreich: Bücherzentrum Meidling, Schönbrunner Straße 261, A-1120 Wien, Tel. 02 22/83 31 96,
Microcomput-ique E. Schiller, Fasangasse 21, A-1030 Wien, Tel. 02 22/78 56 61,
Ueberreuter Media Handels- und Verlagsgesellschaft mbH, Alser Straße 24, A-1091 Wien, Tel. 02 22/48 15 38-0
Bestellungen aus anderen Ländern bitte per Auslandspostanweisung!

Das Angebot dieser Ausgabe:

Programmiersprachen:

Wer sich näher mit Programmiersprachen beschäftigen will, für den haben wir eine Schnupper-Diskette zusammengestellt. Auf ihr finden Sie die für Ihren Computer interessantesten Programme dieser Ausgabe.
Außerdem haben wir unser Archiv durchstöbert. Routinen aus früheren Ausgaben von Happy-Computer, 64er und Computer Persönlich, die in Pascal, C oder Forth geschrieben sind, haben wir gleich mit auf die Diskette geschrieben. Keine Angst, daß Sie diese Programme ohne Beschreibung nicht gebrauchen können. In der Datei »READ.ME« finden Sie die notwendige Beschreibung. Die Schnupper-Diskette gibt es für die Schneider-Computer, den Commodore 64 und den Commodore 128. Im 64er-Modus des C 128 können sie natürlich auch die 64er-Diskette verwenden.

Diskette für Schneider-Computer, Best.-Nr. LH 86S5 SD, DM 34,90* (sFr. 29,50/öS 349,–)
Diskette für C 64, Best.-Nr. LH 86S5 CD, DM 29,90* (sFr. 24,90/öS 299,–)
Diskette für C 128, Best.-Nr. LH 86S5 8D, DM 29,90* (sFr. 24,90/öS 299,–)

Programme aus früheren Ausgaben:

Happy-Computer, Ausgabe 5/86
Commodore 64, Commodore 128
Radish-Two.
Ein Kletter- und Sammelspiel für den C 64.
Ultraboot.
Ergänzung zu »Ultraload Plus«. 104 zusätzliche Blöcke auf der Diskette.
Simple Sound.
Eine kleine Soundbibliothek bietet Klänge für jede Gelegenheit.
Aus Ausgabe 4/86.

Quadrophenia.
Spiel des Monats für den Commodore 64.
Kurven.
Mathematische Kurven auf dem C 128 schnell programmiert. (Läuft nicht im C 64-Modus!)
Kalender.
Ein Kalender für die Jahre bis 2000.
Auto-Boot 128.
Das Programm nutzt die Fähigkeit des C 128, CP/M-Programme automatisch zu booten (laden). (Nicht für C 64).
Widerstände.
Eine Utility, die Ihnen hilft, Widerstandswerte aus Farbskalen in numerische Werte umzurechnen. Aus Ausgabe 5/86.

Diskette für den C 64/C 128
Bestell-Nr. LH 8605 CD
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 4/86
Schneider CPC
D-Mon.
Daten auf Diskette Byte für Byte lesen und ändern. Fehlerhafte Dateien korrigieren und retten.

GOTO XY (nur CPC 464).
Eine mächtige RSX-Befehlserweiterung, die erlaubt, das Ziel von GOTO-GOSUB-Befehlen mit Hilfe einer Variablen zu bestimmen.

Accept.
Ein komfortabler Ersatz für den normalen INPUT-Befehl, mit dem sich jetzt die maximale Eingabe-Länge begrenzen läßt.

Turbo-Screen (nur CPC 464).
Mit dieser RSX-Erweiterung machen Sie der Bildschirmausgabe im Modus 2 Beine
Aus Ausgabe 2/86.

Explora.
Mit diesem Prüfsummen-Generator entfällt die lästige und zeitaufwendige Fehlersuche

Stack-Manipulation (nur CPC 464).
Basic-Programmierung mit vier RSX-Befehlen.
Aus Ausgabe 3/86.

Tool-Basic.
44 neue RSX-Befehle für Grafik-, Sprite-, Disketten- und Kassetten-Programmierung.

Achtes Bit.
Endlich Abhilfe für den Umstand, daß der Schneider CPC über die Drucker-Schnittstelle nur sieben Datenbits ausgibt.

Mord im Computer.
Das DFÜ-Spiel mit Adventure-Charakter.
Aus Ausgabe 4/86.

Best.-Nr. LH 8604 SK (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–
Best.-Nr. LH 8604 SD (Diskette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 3/86
Commodore 64/Commodore 128
Copter-Fight, Husky-Basic, Unser Sonnensystem, Wahlautomat, Softpaint
Bestell-Nr. LH 8603 CD
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 2/86
Commodore 64
Oval Pattern, Börse, Poster Hardcopy, Kassetten-Designer, Super-Sprite, Transbit
Alle 6 Programme auf Diskette für den Commodore 64/128.
Bestell-Nr. LH 8602 CD
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 1/86
Commodore 64/Commodore 128
Taxi. Aus Ausgabe 1/86.
Musik und Farbe. Aus Ausgabe 12/85.
SDB-Sprite Mover. Aus Ausgabe 1/86.
ES-AE. Aus Ausgabe 1/86.
Ultraload. Aus Ausgabe 1/86.
Error 64. Aus Ausgabe 1/86.
Scroll 64. Aus Ausgabe 1/86.
Schatzsuche. Aus Ausgabe 12/85.
SLAD. Aus Ausgabe 12/84.
Alle 9 Programme auf Diskette für den Commodore 64/128
Bestell-Nr. LH 8601 CD
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 12/85
Atari 800XL/130XE/800
Bestell-Nr. LH 8512 B
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 12/85
Schneider CPC
Diskette für den Schneider CPC.
Bestell-Nr. LH 8512 G (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–
Bestell-Nr. LH 8512 D (Diskette)
DM 34,90*/sFr. 29,50/öS 349,–

Happy-Computer, Ausgabe 11/85
Commodore 64
Bestell-Nr. LH 8511 A
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 10/85
Sinclair Spectrum
Bestell-Nr. LH 8510 D
DM 19,90*/sFr. 17,–/öS 199,–
Atari 800XL
Bestell-Nr. LH 8510 B
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 9/85
Commodore 64
Bestell-Nr. LH 8509 A (Diskette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 8/85
Schneider CPC 464
Bestell-Nr. LH 8508 G (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 7/85
Commodore 64
Bestell-Nr. LH 8507 A (Diskette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 6/85
Commodore 64
Bestell-Nr. LH 8506 A (Diskette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 5/85
Schneider CPC 464
Bestell-Nr. LH 8505 G (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 4/85
Commodore 64
Bestell-Nr. LH 8504 A (Diskette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Computer, Ausgabe 3/85
Schneider CPC 464
Bestell-Nr. LH 8503 G (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–

Happy-Sonderhefte

Sonderheft 4/86: Schneider
Bestell-Nr. LH 86S4 K (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–
Bestell-Nr. LH 86S4 D (Diskette)
DM 34,90*/sFr. 29,50/öS 349,–

Sonderheft 3/86: 68000
Bestell-Nr. LH 86S3 D (Diskette)
DM 29,90*/sFr. 24,90/öS 299,–

Sonderheft 2/86: ATARI
Bestell-Nr. LH 86S2 D (2 Disketten)
DM 34,90*/sFr. 29,50/öS 349,–

Sonderheft 1/86: Schneider
Bestell-Nr. LH 86S1 D (Diskette)
DM 34,90*/sFr. 29,50/öS 349,–
Bestell-Nr. LH 86S1 K (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–

Sonderheft 2/85: Schneider
Bestell-Nr. LH 85S2 D (3"-Diskette)
DM 34,90*/sFr. 29,50/öS 349,–
Bestell-Nr. LH 85S2 V (5¼"-Diskette)
DM 34,90*/sFr. 29,50/öS 349,–
Bestell-Nr. LH 85S2 K (Kassette)
DM 29,90*/sFr. 24,90/öS 299,–

Sonderheft 1/85: Spectrum
Bestell-Nr. LH 85S1 D (Kassette)
DM 19,90*/sFr. 17,–/öS 199,–

* inkl. MwSt. Unverbindliche Preisempfehlung

Bitte verwenden Sie für Ihre Bestellung und Überweisung die eingeheftete Postgiro-Zahlkarte, oder senden Sie uns einen Verrechnungs-Scheck mit Ihrer Bestellung.
Sie erleichtern uns die Auftragsabwicklung, und dafür berechnen wir Ihnen keine Versandkosten.

befassen sich daher nationale und internationale Institute. Die wichtigsten sind das Deutsche Institut für Normung (DIN), das American National Standards Institute (ANSI) und die International Organization for Standardization (ISO). Die Benutzer akzeptieren allerdings die Standards unterschiedlich. So sind Standards von Basic nahezu unbekannt, während sie sich bei Sprachen wie Pascal oder Fortran zunehmend durchsetzen.

Was für die Effizienz bei Programmen gesagt wurde, verkehrt sich bei den Sprachen in das genaue Gegenteil: Für den Übersetzer ist sie von großer Wichtigkeit. Zum einen sollte der Übersetzungsvorgang effizient sein, da bei der Fertigstellung eines Programmes die Zahl der Testläufe meistens sehr hoch ist. Der wichtigere Grund aber ist, daß das erzeugte Maschinenprogramm im Hinblick auf Rechenzeit und Speicherbedarf optimiert werden sollte. Sogenannte »Optimierende Übersetzer« sind teilweise in der Lage, das erzeugte Maschinenprogramm effizienter zu gestalten, als dies der Benutzer durch Programmiertricks erreichen kann.

Von Zuse bis Ada

Wir besitzen nun ein gutes Handwerkszeug, um Programmiersprachen nach den wichtigsten Gesichtspunkten theoretisch zu beurteilen. Fassen wir zusammen: Die entscheidenden Anforderungen an eine Sprache heißen Strukturierungshilfen, Selbstdokumentation, Datenbeschreibung, Benutzerfreundlichkeit und Zuverlässigkeit. Doch was nützt all die graue Theorie, wenn wir die Sprachen nicht kennen? Im folgenden werden daher die wichtigsten unter den bisher behandelten Aspekten und im Rahmen ihrer geschichtlichen Entwicklung vorgestellt.

Die Entstehung der Programmiersprachen orientiert sich immer auch an den Voraussetzungen der Hardware. Dies gilt insbesondere auch für die Gründerjahre.

Als geistiger Urvater der Rechenmaschinen mit Programmsteuerung darf der Engländer Charles Babbage gelten. Er begann 1833 mit der Konstruktion digitaler Rechenautomaten. Er legte seinen Maschinen aus Zahnrädern, Kurbeln und Hebeln zwei wichtige Erfindungen zugrunde, nämlich die Lochkartensteuerung und das Prinzip eines dekadischen Zählrades mit automatischem Zehnerübertrag. Babbages Projekte waren aber wegen fertigungstechnischer Schwierigkeiten nur in der Theorie funktionsfähig. Erst Elektromechanik und später die Elektronik machten die Rechenautomaten langsam zu Computern.

Der erste Rechenautomat der Welt mit Programmsteuerung wurde 1941 von Konrad Zuse in Betrieb genommen. Die Zuse Z3 war ein Relaisrechner, der bereits mit Dualzahlen arbeitete und zur Darstellung Gleitkommazahlen benutzte. Mit der Z3 waren neben den vier Grundrechenarten auch das Ziehen von Quadratwurzeln und das Potenzieren möglich. Ein Nachbau des historischen Modells, das im Krieg zerstört wurde, steht im Deutschen Museum in München. Der erste programmierbare Rechner Amerikas entstand 1944. Er wurde von dem Mathematiker Howard H. Aiken mit Unterstützung von IBM entwickelt und auf den Namen »Mark I« getauft. Er war jedoch ein Ungetüm von 16 m Länge und 35 Tonnen Gewicht und zudem langsamer als die früher entwickelte Z3.

Der Phase der Relaisrechner setzte der Einsatz von Elektronenröhren rasch ein Ende. Der bekannte ENIAC war die erste vollelektronische Rechenanlage der Welt und wurde 1945 in den USA fertiggestellt. Er erreichte gegenüber den Relaisrechnern bereits die 2000fache Rechengeschwindigkeit.

Während die Lochkarte nur eine starre Programmsteuerung ermöglichte (keine Schleifen, keine logischen Entscheidungen) begann man bald, sich über die flexible Speicherprogrammierung Gedanken zu machen. Als erstem gelang es dem Amerikaner John Neumann das genannte Problem auf einem Rechner zu verwirklichen. Der bereits 1944 von Neumann konzipierte Computer (EDVAC) erfüllte folgende Forderungen: Das Programm mußte, wie auch die zu verarbeitenden Daten, in der Maschine gespeichert werden. Außerdem benötigte man bedingte Befehle wie Vorwärts- und Rückwärtsverzweigungen. Jeden Befehl konnte zudem die Maschine selbst, wie jeden anderen Operanden ändern.

Befehle bestanden aus einem Operations- und einem Adreßteil. Im Operationsteil wird eine Angabe gemacht was zu tun ist (zum Beispiel Ausführung einer Multiplikation), der Adreßteil zeigt an, wo sich die zu verarbeitenden Daten befinden und wohin sie anschließend zu übertragen sind. Hier ist also die Rede von den ersten Assemblersprachen, an deren Grundprinzipien sich bis heute nicht geändert hat.

Der Schritt von der starren Programmsteuerung zum flexiblen Programm leitete die Wende vom Rechner zur Datenverarbeitung ein. Die Röhrencomputer der ersten Generation erreichten mit dem SSEC (Selective Sequenz Electronic Calculator) Ende der vierziger Jahre einen Höhepunkt. Dieser besaß nicht weniger als 12000 Elektronenröhren und etwa 21500 Relais und wurde von 36 Lochstreifenlesern gesteuert. Er führte die Berechnungen der Mondbahn durch, die 20 Jahre später im Apollo-Raumfahrtprogramm verwertet wurden. Mit dem Einzug der Transistortechnik und später mit den integrierten Schaltkreisen wuchsen fortan Rechnerleistung und Speicherkapazität immer schneller. Dies war die Voraussetzung für die Schaffung der höheren Programmiersprachen.

Die frühen Jahre – Fortran

Fortran ist die älteste der hier behandelten Hochsprachen und setzt einen Meilenstein in der Geschichte. Anfang der fünfziger Jahre wuchs die Zahl der Computer rasch. An Serienfertigung war noch nicht zu denken und es war jedes Gerät ein Einzelstück mit eigener Hardware und eigenem Assembler. So wurde bald der Wunsch nach einer Programmiersprache laut, die übertragbar und einfach zu programmieren sein sollte. 1952 wurde der Grundstein für Fortran gelegt, zu einer Zeit, da die Programmierung nur wenigen Spezialisten und ausschließlich in Assembler möglich war. John W. Backus war einer der Federführenden, dem die Programmiergemeinde Fortran zu verdanken hat.

Der Hauptgrund für die Entwicklung war die Schwerfälligkeit der Assemblerprogrammierung. 75 Prozent der Kosten eines Rechenzentrums verursachte damals die Fehlersuche. Verständlichkeit war daher ein wesentliches Entwurfsziel. Dadurch, daß die teuere Hardware optimal ausgenutzt werden mußte, waren die Rahmenbedingungen für Fortran bereits vorgezeichnet. Vorrangig wurden Sprachelemente implementiert, die der Speicher- und Laufzeiteffizienz nachkamen. Einige dieser Konzepte werden noch heute als sehr nachteilig angesehen – sind aber immer noch in Fortran enthalten.

1955 erschien ein Programmierhandbuch, und zwei Jahre später wurde die erste Implementierung auf einer IBM 704 freigegeben. Damit stand Fortran erstmals einer breiten Zahl von Programmierern zur Verfügung.

Der Name steht für FORmula TRANslating system (Formelübersetzer). Und genau dort liegt auch der Anwendungsschwerpunkt der Sprache. Rechnerische Probleme lassen sich in ihr leicht und natürlich ausdrücken. Damit wird

der Erlernbarkeit der Sprache Rechnung getragen. Im ingenieurwissenschaftlichen und mathematischen Bereich gilt Fortran auch heute noch als die wichtigste Programmiersprache. So bietet sie zum Beispiel neben den allgemein gebräuchlichen Zahlentypen Real (Fließkommazahlen) und Integer (Ganze Zahlen) auch noch den Typ »Double Precision« für Rechnungen mit höherer Genauigkeit sowie »Logical« für boolsche Operationen. Großrechner-Versionen beinhalten zudem noch den Typ »Complex«, der in der theoretischen Elektrotechnik eine sehr wichtige Rolle spielt.

Das Format dieser Sprache ist streng zeilenorientiert und erlaubt normalerweise nur einen Befehl je Zeile. Das hängt damit zusammen, daß Fortran zunächst als lochkartenorientierte Sprache entstand. Grundsätzlich mußte man damals für jede neue Anweisung eine neue Lochkarte (beziehungsweise Zeile) verwenden. Wie auch in Basic, das später aus Fortran entstand, mußte bei den ersten Versionen viel mit dem Goto-Befehl umhergesprungen werden. Neuere Versionen wie Fortran V und Fortran 77 bieten demgegenüber schon strukturierende Sprachelemente wie IF....THEN....ELSE...ENDIF.

Nachdem die 1958 geschaffene Version Fortran II eine mäßige Verbreitung gefunden hatte, entstand 1962 das in weiten Kreisen akzeptierte Fortran IV. Den fortschreitenden Auswüchsen immer neuer Versionen wurde 1966 Einhalt geboten, mit einer Version, die größtenteils mit Fortran IV identisch war. Schließlich überarbeitete das ANSI Fortran 66 im Jahr 1977 nochmals und beseitigte einige eklatante Mängel.

Unter CP/M ist Fortran derzeit für fast alle Mikrocomputer mit Z80-Prozessor erhältlich, ebenso wie eine Reihe von Fortran-Implementationen für MS-DOS-Computer.

Cobol – die Geschäftige

Die Programmiersprache Cobol entstand 1959 auf Initiative des US-Verteidigungsministeriums. Zu dieser Zeit begann Fortran sich gerade auszubreiten. Was noch fehlte, war eine Sprache für den kommerziellen und kaufmännischen Einsatz. So entwickelte man Cobol mit dem Ziel, große Datenbestände verarbeiten zu können und die Ein/-Ausgabe zu unterstützen. Insbesondere die ersten Fortran-Versionen waren hierfür ungeeignet. Ende der fünfziger Jahre wurde die Codasyl-Entwicklungsgruppe aus Vertretern der Computerindustrie und der amerikanischen Regierung gegründet. Schon 1960 stellte diese Gruppe die erste Version mit der Bezeichnung Cobol-60 vor. Sie war wesentlich an die weniger bekannte Sprache Comtran (COMmercial TRANslator) angelehnt. Aufgrund der hastigen Entwicklung von Cobol innerhalb eines halben Jahres ergaben sich viele Ungereimtheiten, die sich teilweise durch alle Neuentwürfe hindurchschleppten und auch heute noch nicht ganz beseitigt sind. Cobol-61 war dann die Grundlage für alle späteren Versionen. Sie war zu Cobol-60 nicht kompatibel. 1965 wurde als wesentliche Neuerung die Unterstützung von Massenspeichern und Tabellen mit eingebracht.

Die Sprachelemente von Cobol sind je nach ihrer Funktion in Module zusammengefaßt. Das ANSI entwickelte bis 1974 einen zwölf Module umfassenden Standard, namentlich Cobol ANS-74. Er wurde 1980 nochmals verbessert. Den jeweils neuesten Stand veröffentlicht das CODASYL-Komitee im Abstand von drei Jahren. Der Standard für die Bundesrepublik Deutschland ist in der DIN-Norm 66028 nachzulesen.

Die kurze Darstellung der Entwicklungsgeschichte läßt erkennen, daß Cobol ebenfalls eine alte Sprache ist. Cobol-Programme müssen aus heutiger Sicht als mangelhaft angesehen werden.

Ursprünglich verfolgte man wie bei keiner anderen Sprache das Ziel der Lesbarkeit des Programmtextes. Die Sprache sollte dann auch leicht erlernbar sein. Der Programmtext erinnert stark an (englische) Prosa. Cobol-Programme simulieren nämlich die natürliche englische Sprache. So wird zum Beispiel jeder Befehl mit einem Verb eingeleitet. Die Grundrechenarten stehen nicht als Symbole, sondern als Befehlswörter (add, divide) zur Verfügung. Ebenso werden logische Operatoren ausgeschrieben. Ein Beispiel: Wenn die Variable A größer ist als Null, soll der Variablen B die Summe der Variablen C und A zugeordnet werden. In Basic würde man das so formulieren:

```
IF A>0 THEN B= C+A
```

Daraus wird in Cobol:

```
IF A GREATER THAN ZERO ADD C TO
A GIVING B.
```

Daß so aus komplizierteren mathematischen Formeln monströse Gebilde werden, leuchtet ein. Da höhere mathematische Funktionen in Cobol ganz fehlen, ist die Sprache für wissenschaftliche Anwendungen völlig ungeeignet.

Derartige Beispiele verdeutlichen, daß das Entwicklungsziel von Cobol nicht sinnvoll erreicht wurde. Dennoch ist Cobol auch heute noch die weltweit am stärksten verbreitete Programmiersprache. Ganze Rechenzentren arbeiten mit ihr. Die hohen Investitionen und die Gewöhnung des Personals an diese Sprache wird auch in Zukunft für ihren Erhalt sorgen. Zudem erreicht auch noch keine neuere Programmiersprache die Cobol-Domäne Datenorganisation.

Wer einen Mikrocomputer mit den Betriebssystemen CP/M oder MS-DOS (PC-DOS) besitzt, kann in Cobol einsteigen.

PL/1 – von allem etwas

Fortran und Cobol sind charakteristisch für die strikte Trennung in kommerzielle und technisch-wissenschaftliche Anwendungen zu Beginn der sechziger Jahre. Daneben wurden Computer nur noch in Spezialgebieten eingesetzt. Im Laufe der Zeit traten aber die Merkmale der naturwissenschaftlich-technischen Bereiche in den betriebswirtschaftlichen Anwendungen immer mehr hervor und umgekehrt. So waren die kommerziellen Anwender mehr und mehr auf Methoden aus der Statistik, Operations Research und Ökonomie angewiesen. Mathematiker und Ingenieure stellten zunehmend höhere Ansprüche an Datenverwaltung und an die Ein-/Ausgabeunterstützung.

Als logische Konsequenz kamen die Anbieter den neuen Bedürfnissen mit einer universellen Hard- und Softwarekonfiguration nach. Hardwareseitig entwickelte IBM die Rechnerfamilie /360 mit dem Betriebssystem OS/360. Diese Anlagen zählten sich bereits zur dritten Computergeneration (das heißt, die Schaltkreise waren in Hybridtechnik ausgelegt, einer unmittelbaren Vorstufe der integrierten Schaltkreise).

Bei den Überlegungen zu einer neuen Sprache war man zunächst von Fortran ausgegangen. Die Organisation SHARE (Society for Help to Avert Redundant Effort), eine Vereinigung wissenschaftlicher IBM-Anwender, einigte sich mit der Firma IBM auf die Gründung eines Sprachkomitees. Zunächst war die Rede von Fortran VI. Man gelangte aber schon nach kurzer Zeit zu der Erkenntnis, daß die gewünschten Verbesserungen eine Kompatibilität mit Fortran unmöglich machten. Die Anlehnung an Fortran hätte außerdem die große Gruppe der kommerziellen Verwender abgeschreckt. So entschied man sich für die Entwicklung einer gänzlich neuen Sprache. Deren wichtigsten Entwurfsprinzipien waren: allgemeine Einsetzbarkeit und weitgehende Ausdrucksfreiheit. Der Sprachaufbau sollte modular sein und Testhilfen sowie Möglichkeiten zur

Fehlerbehandlung bieten. Ein großenteils gegenläufiges Ziel bestand in den Forderungen, den vollen Zugriff auf Hardware- und Betriebssystemleistungen bei gleichzeitiger Maschinenunabhängigkeit zu gewähren.

Nach vielen drastischen Überarbeitungen hatte sich der Sprachumfang bis 1965 stabilisiert. Nachdem die Abkürzung für NPL (New Programming Language) bereits vergeben war, einigte man sich schließlich auf PL/I (Programming Language one). Im August 1966 wurde dann der erste Compiler für eine IBM /360 freigegeben. Schließlich verabschiedeten das ANSI und die ECMA, ein europäisches Standardisierungskomitee, einen vorläufig endgültigen Standard. Die Verbreitung von PL/1 nahm zunächst rasch zu, wurde aber später den gesetzten Erwartungen nicht gerecht.

Kommen wir nun zu der Sprache, die jeder, und wenn nur vom Hörensagen, kennt. Sie kann sich rein zahlenmäßig im Mikrocomputerbereich als die am

In den Sprachumfang von PL/1 wurden viele Konzepte aus Fortran, Cobol, Algol und Jovial übernommen (letztere werden hier wegen ihrer geringen Verbreitung nicht behandelt). Leider gelang es nicht, sich bei der Auswahl nur auf die guten Eigenschaften der Vorgänger zu beschränken. Zudem ist der Sprachumfang riesig. Eine fast unüberschaubare Anzahl von Schlüsselwörtern zwingt zu Maßnahmen, durch die der Programmierer auch mit Teilmengen der Sprache sinnvoll arbeiten kann. Daher ergeben sich für die PL/1 -Syntax sehr freizügige Vorschriften. Einerseits existiert keine Zeilenstruktur, Zwischenräume, Einrücken und Kommentare dürfen fast beliebig verstreut werden. Da passiert es dann auch nicht selten, daß beim Programmieren das Format aus allen Fugen gerät. Außerdem sind die Schlüsselwörter nicht reserviert, wohl wegen ihrer großen Anzahl. Gebilde wie

```
IF ELSE=THEN THEN IF=ELSE; ELSE
IF=THEN
```

tragen wohl zur Verwirrung jedes hoffnungsvollen Programmierers bei. Ein wesentlich angenehmerer Fortschritt ist andererseits das ausgeprägte Blockkonzept der Sprache. PL/1 orientiert sich wesentlich an Prozeduren. Allgemein kann gesagt werden, daß die Einarbeitung in PL/1 viel Zeit braucht. Wer damit trotz aller Warnungen beginnen will, kann dies unter CP/M oder auf PC-Kompatiblen tun.

weitesten verbreitete Sprache der Welt rühmen. Sie ist schon fast als Teil der Allgemeinbildung zu betrachten. Ganz anders als bei den »großen professionellen« wurde die Entwicklung von Basic (Beginners All-purpose Symbolic



Instruction Code) nicht durch eine Industrie- oder Militärlobby getragen. Die Ausrichtung der Sprache kommt denn auch in ihrem Namen zum Ausdruck: Sie wendet sich an den Anfänger und soll für jeden Zweck geeignet sein.

Basic – Basis für Einsteiger

Ihre geschichtliche Entwicklung erklärt viele Aspekte des Sprachkonzepts. Basic wurde von Thomas Kurtz und John Kemeny von 1956 bis 1971 in den USA am Dartmouth College entwickelt. Ziel war es, Studenten, die sich nicht ausschließlich mit Ingenieurwissenschaften beschäftigten, das Programmieren zu erleichtern. So schlugen sich denn auch die Erfordernisse der allgemeinen Ausbildung einer Universität in der Sprache nieder: Bei der angesprochenen Zielgruppe erschien ein eigener Programmierkurs nicht erforderlich. Vielmehr sollte das Programmieren im Rahmen der Mathematikvorlesungen gelehrt werden. Hieraus erklärt sich auch die Ausrichtung von Basic auf mathematische Probleme. Die neue Sprache sollte leicht erlernbar und leicht zu benutzen sein. Dies war nach Meinung von Kurtz und Kemeny bei Fortran und Algol nicht der Fall. So erklärt sich auch, daß bei Basic nicht, wie bei PL/1, auf Bewährtes zurückgegriffen wurde.

Im Gegensatz zu den bisher behandelten Sprachen wurde Basic als vollständiges Programmiersystem konzipiert. Der Benutzer kann im Dialog mit Basic arbeiten, ohne die Basic-Umgebung zu verlassen. Hierzu existiert der »Direkt-Modus«, der das Editieren, Ausführen und Speichern von Programmen unterstützt. Basic ist zudem eine typische Interpreter-Sprache, für die allerdings auch verschiedene Compiler erhältlich sind.

Mit der Entwicklung des ersten Compilers begannen Kurtz, Kemny und eine Gruppe Studenten 1963. Im Mai 1964 wurde dann das erste Basic-Programm ausgeführt; die erste Version kannte nur 14 Instruktionen. Diese Minimalausstattung wurde am Dartmouth College bis 1971 in insgesamt sechs Versionen schrittweise vervollständigt. Seither nahmen die Autoren keine Veränderungen mehr vor. Das bedeutet natürlich nicht, daß Basic jemals eine wirkliche Standardisierung erfahren hätte. Im Gegenteil: Durch das Fehlen einer Interessenvertretung professioneller Anwender und durch die enorme Verbreitung der Sprache wucherten die Basic-Versionen fast uferlos. Für nahezu jeden Mini- und Mikrocomputer und selbst auf Taschenrechnern ist Basic erhältlich. Da von Herstellerseite nach dem Motto verfahren wurde »jedem Topf ein anderer Deckel« ist Kompatibilität ein Fremdwort. Auch die Bemühungen der ECMA und ANSI in den letzten Jahren waren kaum von Erfolg gekrönt. Lediglich ein »Minimal Basic« wurde kompromißbereit zum Standard deklariert. Diese Teilmenge ist der ersten Sprachbeschreibung von 1964 sehr ähnlich. Lediglich im Heimcomputerbereich gelang es mit den MSX-Computern erstmals, einen weitestgehenden Basic-Standard für Geräte verschiedener Hersteller zu schaffen. Haar in der Suppe war aber, daß die MSX-Computer wenig Verbreitung fanden.

Ein Basic-Programm besteht aus nummerierten Zeilen in aufsteigender Folge. Dabei sind die Zeilennummern aus einem festgelegten Intervall zu wählen. Jeder Zeilennummer folgen eine oder mehrere Instruktionen. Die

Zeilennummern legen die logische Reihenfolge der auszuführenden Anweisungen fest. Darüber hinaus dienen sie als Orientierung für die Sprungbefehle. Beim Editieren des Programmtextes lokalisieren sie die Zeilen. Basic ist noch stärker zeilenorientiert als Fortran. Das geht so weit, daß eine Basic-Anwendung durch die Zeilenlänge begrenzt wird. In einigen Basic-Versionen sind Fortsetzungszeilen mit dem Zeichen »&« möglich.

Als Datentypen sind in Basic nur numerische Daten und Zeichenketten vorgesehen. Eine Unterscheidung in ganzzahlig und Fließkomma, wie es von anderen Sprachen her bekannt war, wurde der Einfachheit wegen bewußt vermieden, ist aber dennoch auf vielen Computern implementiert. Für Datenstrukturen stehen nur ein- und mehrdimensionale Felder (Arrays) zur Verfügung. Deren Elemente können je nach Version entweder nur einzeln manipuliert werden, oder es stehen spezielle Matrizenoperationen bereit.

Die Sprunganweisungen sind, wie bereits erwähnt, vielen ein Greuel. Zwar bieten moderne Basic-Versionen viele Befehle, die das strukturierte Programmieren unterstützen, sie erfordern aber genaue Vorausplanung eines Programms. Im Regelfall endet bei längeren Programmen ein »Drauflosprogrammieren« im Chaos aus GOTO, GOSUB und FOR...NEXT.

Des weiteren beinhaltet Basic, je nach Ausstattung und Computer, Befehle für die Ein-/Ausgabe, mathematische Funktionen, Grafik, Tonerzeugung und den Zugriff auf das Betriebssystem. Hierauf im einzelnen einzugehen würde zu weit führen. Es sei daher auf entsprechende Fachliteratur verwiesen.

Pascal - strukturiert und einfach

Ein moderner Klassiker unter den Programmiersprachen ist Pascal. Der Name ist ausnahmsweise keine Abkürzung. Er wurde zu Ehren des französischen Mathematikers Blaise Pascal gewählt, der 1642 im Alter von 19 eine der ersten funktionsfähigen Rechenmaschinen konstruierte.

Die Wurzeln dieser Programmiersprache reichen in das Ende der sechziger Jahre zurück. Zu jener Zeit stellten Nikolaus Wirth und C.A.R. Hoare Überlegungen an, auf der Basis von Algol 60 eine Nachfolgesprache zu entwickeln. Algol 60 bot schon damals ein zufriedenstellendes Sprachkonzept. Das gilt besonders für die Strukturierung des Programmablaufs und des Programmtextes. Wegen dieser Vorteile wurde Algol zur Grundlage einer ganzen Klasse von Programmiersprachen, die bei Pascal beginnt und vorläufig von Ada gekrönt wird. Eine der Schwächen ist dagegen die unzureichende Datenstrukturierung. Ebenso wie Basic und Fortran kennt Algol 60 nur das Array. Die später folgende Version Algol 68 war ähnlich wie PL/1 zu umfangreich und unhandlich. Mit der Entwicklung von Pascal verfolgte man das entgegengesetzte Ziel. Der Schweizer Professor Nikolaus Wirth formulierte bei der Entwicklung der Sprache an der ETH Zürich die folgenden Schwerpunkte: Pascal sollte nur grundlegende Sprachkonzepte enthalten. Diese sollten natürlich definiert sein und das Erlernen des strukturierten Programmierens als eine systematische Disziplin unterstützen. Des weiteren sollte sie sich effizient auf allen Computern implementieren lassen.

Die erste vorläufige Version entstand 1968. Die vollständige Beschreibung eines Compilers und der Sprache selbst war 1971 fertig. Das 1974 erschienene Benutzerhandbuch »Pascal User Manual and Report« enthält eine Sprachdefinition, die heute als Wirth-Standard bezeichnet wird. Die verbreiteten Versionen Turbo- und UCSD-Pascal enthalten demgegenüber noch einige Erweiterungen, die vor allem Grafik und Zeichen-Strings betreffen.

Durch die leichte Erweiterbarkeit von Pascal entstanden bald viele Dialekte. Deshalb und wegen der weltweiten Anerkennung der Sprache nahmen sich verschiedene Normeninstitute diesem Problem an. Die ISO setzte schließlich 1980 einen Standard fest, der in der DIN-Norm 66256 nachzulesen ist. Strukturierung bedeutet nicht nur, daß das Programm übersichtlich ist, sondern daß sich der Vorgang des Programmierens in verschiedene Aktionen aufteilt. Bevor man sich an den Computer setzt, sollte man das Problem genau analysieren und in Aufgabenpakete zerlegen. Dann ist für jedes Paket ein Algorithmus zu bestimmen und in Pascal zu formulieren. Erst dann beginnt die Tipparbeit. Pascal-Programme entstehen also auf dem Papier und weniger am Bildschirm. Diese Vorgehensweise erreicht eine niedrige Fehlerquote und damit sinkt auch die Zahl der Übersetzerdurchläufe, was bei einer typischen Compilersprache wie Pascal sehr angenehm ist.

Ein Pascal-Programm ist klar gegliedert in einen Vereinbarungs- und einen Anweisungsteil. Vereinbart werden zuerst alle Variablen, Konstanten und deren Datentypen. Im Anweisungsteil werden die Aufgabenpakete in Procedures formuliert. Jede Procedure enthält einen eigenen Namen. Das eigentliche Hauptprogramm besteht dann nur noch aus dem Aufrufen der Procedures und steht im Programmtext ganz am Ende.

Daß der GOTO-Befehl in Pascal enthalten ist, verwundert eigentlich. Angesichts der Strukturbefehle IF....THEN-CASE kann man gut auf ihn verzichten. Der Vorrat an Datentypen eröffnet gegenüber Basic ganz neue Möglichkeiten. Man unterscheidet hier zwischen einfachen Typen, strukturierten Typen und Zeigertypen. Integer, Char, Boolean und Real zählen zu den einfachen Typen. Boolean bezeichnet eine Variable, der nur die Werte False oder True zugeordnet werden können. Array, Record, Set und File stehen als strukturierte Typen zur Verfügung. Set bezeichnet eine Menge. Auf diesen Typ sind die üblichen Mengenoperationen Vereinigung, Durchschnitt, Differenz, Untermenge und Elementüberprüfungen anwendbar. Record ermöglicht Verbundvariablen. Es lassen sich so unterschiedliche Variablentypen zu einer Variable zusammenfassen. Record war ursprünglich für kommerzielle Anwendungen gedacht (Tabellendarstellung). Demgegenüber werden mit dem Typ File nur Variablen eines einzigen Typs verkettet. Der Typ Zeiger (pointer) schließlich ermöglicht verkettete Listen und deren bequeme Manipulation, sowie Baumstrukturen. Wem das noch nicht reicht, der kann sich in Pascal weitere einfache Datentypen selbst definieren.

Pascal ist vielseitig und erzieht zum strukturierten Denken. Seine Verarbeitung, vor allem im akademischen Bereich, ist folgerichtig sehr hoch. So liegen denn auch für fast alle rechnereigenen Betriebssysteme wie auch für CP/M und MS-DOS Pascal-Versionen vor.

Forth — die etwas andere Sprache

Forth entstand Anfang der siebziger Jahre. Charles H. Moore entwickelte die Sprache ursprünglich zur Steuerung von Radioteleskopen. Er arbeitete dazu mit einem IBM-1130, einem Computer der dritten Generation. Das Endprodukt war aber so mächtig, daß es Moores Computer als einen der vierten Generation erscheinen ließ. Er wollte der neuen Sprache daher den Namen Fourth geben. Namen mit mehr als fünf Buchstaben waren auf dem IBM jedoch nicht erlaubt. So wurde das »u« ein Opfer dieser technischen Unzulänglichkeit.

Forth ist interaktiv wie Basic. Das heißt, es existiert sowohl ein Interpreter als auch ein Compiler. Programme können somit erst im Direktmodus »häppchenweise« getestet und anschließend compiliert werden. Des weiteren verbindet Forth Merkmale der Assemblersprachen mit denen der Hochsprachen.

Die Strukturierung in Forth entsteht durch die Definition immer neuer Worte. Der ohnehin schon große Sprachumfang nimmt beim Programmieren ständig zu. Beliebige Befehle können zu einem neuen Befehl zusammengefaßt werden, der dann sofort wieder in weitere Befehle mit eingebaut werden kann. Schließlich steht für das gesamte Programm ein einziger Befehl am Ende dieser Kette.

Selbstverständlich stellt Forth auch die Kontrollstrukturen zur Verfügung, die bereits für Pascal angesprochen wurden, wie IF.....ELSE....ENDIF, BEGIN...UNTIL, BEGIN...WHILE etc. Das berüchtigte GOTO fehlt hier ganz, ergäbe auch bei diesem Sprachkonzept keinen Sinn.

Grundlegendes Prinzip von Forth ist das Operieren mit dem Stapelspeicher (Stack). Dieser funktioniert nach dem LIFO-Prinzip (Last In, First Out). Alle Werte, die auf dem Stapel abgelegt wurden, lassen sich nur in umgekehrter Reihenfolge wieder herunternehmen. Für einen problemlosen Ablauf dieses Systems sorgen eine ganze Reihe von Stack-Befehlen, mit denen sich Werte verschieben und vertauschen lassen. Sämtliche mathematischen Operationen laufen in Forth über den Stack. Man bedient sich hierbei der Umgekehrten Polnischen Notation (UPN), die recht gewöhnungsbedürftig und Benutzern von HP-Taschenrechnern bekannt ist.

Forth ist ein sehr offenes System und durch seine Assemblernähe universell einsetzbar. Fehlende Funktionen können jederzeit selbst programmiert werden. Zudem ist Forth sehr schnell. Das Erlernen der Sprache und die Übersichtlichkeit der Programme können als noch ausreichend eingestuft werden. Auf jeden Fall fasziniert Forth jeden, der sich länger damit beschäftigt. Für alle verbreiteten Mikrocomputer existieren mittlerweile eine oder mehrere Versionen.

Logo — kinderleicht

Seymour Papert, der als geistiger Vater der Sprache Logo gilt, arbeitete 12 Jahre an der Verwirklichung dieser »Erziehungsphilosophie«. Er leitete ein eigens gegründetes Entwicklungsteam aus Programmierern und Lehrkräften am MIT (Massachusetts Institute of Technology) in den USA. Man arbeitete damals ausschließlich auf den größten vorhandenen Datenverarbeitungsanlagen. Dadurch fand ein wesentliches Konzept der KI-Sprache Lisp in Logo Anwendung: die Listenprogrammierung. Listen sind einfach zu definieren und können per Befehl manipuliert, kombiniert und verglichen werden. Eine leicht programmierbare Dateiverwaltung ist nur ein Anwendungsbeispiel dieser Technik.

Bekannt wurde Logo vor allem durch die Schildkröte, ein kleines Zeichensymbol der »Turtle-Graphics«. Mit ihr lassen sich auf einfache Weise die tollsten Grafiken zaubern. Die Schildkröte krabbelt über den Bildschirm und hinterläßt dabei eine sichtbare Spur.

Eine Schildkröte machte Logo bekannt

Gesteuert wird mit einfachen Befehlen wie FORWARD, BACK, LEFTTURN, RIGHTTURN. Zusätzlich muß noch die Länge der zurückgelegten Strecke und des Drehwinkels angegeben werden. Ebenso ist eine Standortabfrage der Turtle möglich. Logo-Programme ähneln in der Struktur dem Baukastenprinzip der Forth-Programme. Mit Hilfe des Interpreters lassen sich einzelne Bausteine erproben und später zum eigentlichen Programm zusammensetzen. Der Komfort ist dabei in Logo ungleich höher als in allen bisher genannten Sprachen. So kann der Anwender vorerst Begriffe wie Archivierung, Dateien und andere spezielle Funktionen der Datenverarbeitung links liegen lassen. Diese Vorteile gehen aber leider zu Lasten des Speicherplatzes.

Eng mit dem Prozedurkonzept verbunden ist die Rekursivität von Logo. Prozeduren sind in der Lage, sich selbst aufzurufen. Auf diese Weise lassen sich schnell reizvolle grafische Gebilde erzeugen und gewisse mathematische Zusammenhänge einfach ausdrücken. Rekursive Strukturen sind in Basic gar nicht und in vielen anderen Sprachen nur mit hohem Aufwand zu verwirklichen.

In die Logo-Philosophie wurden Erziehungstheorien des Schweizer Philosophen Jean Piaget eingebracht. Dieser hatte zuvor das Lernverhalten von Kindern analysiert. Tatsächlich wirkt Logo besser auf die Denkweise eines Schülers als Basic oder Pascal. Bemängelt werden muß bei Logo hauptsächlich die geringe Verarbeitungsgeschwindigkeit der Programme. Sie fällt aber bei einem Lernsystem nicht so stark ins Gewicht.

Wegen des hohen Speicherbedarfs sind Logo-Interpreter auf Mikrocomputern nur als Ausschnitt des Gesamtsystems erhältlich. Dies wird sich mit wachsendem Speicherstandard jedoch bald ändern.

Comal — gelungene Essenz

Im Jahre 1973 ging Comal aus den Sprachen Basic und Pascal als ein neuer Ableger hervor. Später kamen in Comal noch Elemente von Logo hinzu, so zum Beispiel die Schildkrötengrafik. Zudem sind in Comal der Compiler und der Interpreter nicht getrennt vorhanden, sondern es wurden deren beste Elemente in einer Zwischenstufe zusammengefaßt. Ein Comal-Programm besteht aus drei Schritten. Im ersten wird schon bei der Programmeingabe die Syntax überprüft. Dieser Syntaxchecker ist selbst für den ohnehin schon eingabefreundlichen Interpreter ungewöhnlich komfortabel. Die Comal-Schlüsselworte werden sofort in sogenannte »Token« übersetzt, das sind Abkürzungen, die nur ein Byte beanspruchen. Dieses Prinzip verwenden übrigens alle Interpreter. Der zweite Schritt beginnt nach dem Programmstart. In einer Art Compilerdurchlauf wird der Programmtext nach Variablen, Prozeduren, Funktionen und Sprüngen durchsucht. Die Ergebnisse dieser Analyse werden dann in einer gesonderten Liste zusammengefaßt. Man kann diesen Vorgang auch als eine Art automatische Deklaration ansehen. Im dritten Schritt, dem Programmlauf selbst, wird auf diese Liste ständig direkt zugegriffen. So ergeben sich gegenüber Basic, wo der Interpreter oft den ganzen Programmtext absucht, enorme Geschwindigkeitsvorteile.

Die Comal-Syntax lehnt sich stark an die von Basic an. Im Direktmodus wurden die meisten Befehle Wort für Wort übernommen. Das gleiche gilt für einige Befehle des Programm-Modus. Einige Comal-Versionen akzeptieren neben den eigenen Schlüsselworten sogar noch gleichbedeutende Basic-Befehle. Wer aus Basic zu Comal aufsteigt, ist so zwar vor Irrtümern einigermaßen sicher, jedoch wird dem Prinzip der Eindeutigkeit nicht gerade Rechnung getragen.

Von Pascal wurde die Strukturiertheit übernommen, dessen strenge Syntaxvorschriften aber erfreulicherweise vermieden. Die typischen Kontrollstrukturen, die schon bei Pascal und Basic beschrieben wurden, sind ausnahmslos vorhanden. Die Lesbarkeit des Pro-

grammtextes wird zudem dadurch unterstützt, daß Comal beim Listen eine optische Gliederung vornimmt.

Comal wird für fast alle gängigen Mikrocomputer angeboten. Erfreulicherweise stehen auch, ähnlich wie bei Forth, zahlreiche Public-Domain-Versionen zur Verfügung. Diese unterscheiden sich von kommerziellen Sprachangeboten lediglich im Umfang. Es wird erwartet, daß Comal in Zukunft eine starke Verbreitung erfährt. Jüngster Anhaltspunkt für diese These ist der Beschluß der Kultusministerien, Pascal im Informatikunterricht allmählich durch Comal zu ersetzen.

C – die Zukunft?

»C« – für diesen einen Buchstaben lassen viele Programmierer Pascal, Forth oder Fortran links liegen. Viele betrachten C als die Programmiersprache schlechthin. Tatsächlich bietet C viele bestechende Vorteile, die sich von denen der anderen Hochsprachen unterscheiden. In C wurden bekannte Betriebssysteme wie Unix und GEM geschrieben.

Die Entwicklung von C reicht bis an den Anfang der siebziger Jahre zurück. Die Namensgebung war ebenso kurios wie einfallslos. 1970 begann die Firma Digital Equipments mit der Entwicklung von Spezialsprachen, die die Programmierung von Minicomputern unterstützen sollten. Diese wurden einfach nach dem Alphabet benannt. B war also eine Weiterentwicklung von A und diente 1971 dazu, Unix auf verschiedene Rechner zu übertragen. Bald darauf erkannten Dennis Ritchie und Ken Thompson, die damals bei Bell Laboratories beschäftigt waren, die Leistungsfähigkeit von B. Sie verbesserten diese Sprache bis 1973 entscheidend. Wie das Endprodukt heißt, wissen wir bereits. Unix wurde wenig später ebenfalls auf C umgeschrieben.

Wenn ganze Betriebssysteme in C gehalten sind, läßt sich die ungeheuere Effizienz dieser Sprache schon erahnen. Das C-Compilat geht mit dem Speicher äußerst sparsam um, und ist zudem um den Faktor 50 schneller als vergleichbare Basic-Programme.

Der eigentliche Befehlsvorrat von C ist denkbar klein. Er umfaßt nur 13 Instruktionen. Bei der Erzeugung eines C-Programms wird der Quelltext zuerst mit dem mitgelieferten Editor oder einer Textverarbeitung geschrieben. Dieser dient dann als Eingabedatei für den Compiler. Bis hierher besteht also kein Unterschied zur Vorgehensweise mit den meisten anderen Compilersprachen. Der Compiler überprüft den Text und übersetzt diesen mit den ihm bekannten Befehlen in eine Zwischendatei und legt diese auf einen externen Speicher (zum Beispiel Diskette) ab. Anschließend beginnt das sogenannte »Linking«. Der Linker ordnet aus einer externen Bibliothek die noch fehlenden Befehle entsprechendem Assemblercode zu. So entsteht schließlich das lauffähige Programm, das fast so kompakt ist, als wäre es direkt in Maschinensprache geschrieben. Das Compilieren benötigt jedoch wegen des Zwischencodes je nach Geschwindigkeit des Speichermediums mehr Zeit, als die unmittelbare Übersetzung in Maschinensprache.

Die Vorteile des Compiler-Linker-Prinzips wiegen diesen Nachteil bei weitem auf: C ist praktisch uneingeschränkt portabel und die Linker-Funktion kann man selbst erweitern. C-Programme bieten neben der selbstverständlichen Strukturierung einige ungewöhnliche aber vorteilhafte Eigenschaften. So sind in Anlehnung an Assembler Befehle zum Inkrementieren und Dekrementieren vorhanden, Variable können als Registervariable deklariert werden. Daß die Benutzung derartiger Sprachmittel einem Compiler das Leben leicht macht, ist klar. Einschränkungen entstehen jedoch bei Prozessoren, die nur wenige Register aufweisen, wie beispielsweise die drei 8-Bit-Register des 6502, der nur A, X und Y zu bieten hat.

Ein weiteres nennenswertes Stilmittel unter C sind Makros. Diese sind mit Unterprogrammen vergleichbar, die nicht angesprungen werden müssen, sondern jeweils erneut vom Compiler in den Objektcode eingebunden werden. Diese Technik bringt einen großen Zeitvorteil, da Parameterübergaben und Sprünge entfallen.

Im 8-Bit-Bereich ist C nur auf dem Z80 unter CP/M verbreitet. Eine Version für den C 64 stellt hier eine erfreuliche Ausnahme dar. Für 16- und 32-Bitter existieren aber umfangreichere Versionen.

Ada – gekrönter Adel

Ada ist heute als Krönung bei der Entwicklung modularer Programmiersprachen anzusehen. Die Sprache wurde erst in jüngster Zeit im Auftrag des weltweit größten Softwaresponsors entwickelt, dem Pentagon. Der finanzielle und organisatorische Aufwand dafür war entsprechend riesig. Benannt ist die Sprache nach der jungen Gräfin Ada Byron, die um 1830 für Babbages (siehe oben) Rechenmaschinen ein nahezu komplettes Programm zur Berechnung der Bernoullischen Zahlen schrieb.

Der Aufwand, der um Ada getrieben wurde, erklärt sich mit einer Kalkulation des US-Verteidigungsministeriums. Danach können zwischen 1983 und 1999 etwa 24 Milliarden Dollar (!) eingespart werden, wenn eine einzige universelle Programmiersprache die bisherigen 450 (!!) Programmiersprachen ersetzen könnte. Ada ist ähnlich PL/1 sehr umfangreich. Es wäre daher zwecklos, auf einzelne Sprachelemente einzugehen. Deshalb hier nur die grundsätzlichen Sprachkonzepte von Ada:

- Das Modulkonzept von Ada ist äußerst umfangreich. Es stehen sowohl datenorientierte als auch funktionsorientierte Module zur Verfügung. Innerhalb der datenorientierten Pakete lassen sich fast beliebige Datentypen und Datenstrukturen realisieren.
- Ähnlich wie in Pascal können Datenstrukturen geschachtelt werden. Umfangreichere Prozedur- und Funktionskörper werden ausgelagert, zum Beispiel um die Lesbarkeit der Programme zu erhöhen.
- Sämtliche Kontrollstrukturen, von UNTIL bis hin zu CYCLE-Schleifen stehen zur Verfügung. Ferner sind alle linearen und strukturierten Datentypen implementiert.
- Ein automatischer Textformatierer (Pretty-Printer) wertet Schachtelungsstrukturen aus und sorgt für ein übersichtliches Layout des Programmtextes.
- Neben Parallelverarbeitung (Multitasking) gehören Konzepte wie die Parallel- und Ausnahmebehandlung zu den bemerkenswerten Fähigkeiten, auf deren Erklärung hier wegen der komplexen Zusammenhänge verzichtet wird.

Im großen und ganzen wurden die gesetzten Ziele bei der Entwicklung von Ada nach dem heutigen Erkenntnisstand optimal erreicht. Die zu Anfang unter den Qualitätsaspekten genannten Stichworte wie Vollständigkeit, Zuverlässigkeit, Korrektheit, Übertragbarkeit, Wartung und Fehlerbehandlung wurden weitestgehend realisiert. Die Einfachheit der Sprache ist als ausreichend zu betrachten, wenn auch die vollständige Einarbeitung in dieses System Jahre beansprucht. Ada ist für Mikrocomputer zur Zeit nur unter MS-DOS verfügbar und auch hier nur in einer abgespeckten Version.

Wir sind nun mit dem Aufstieg im »modernen Turm zu Babel« fast an der Spitze angelangt. Nach unten blickend können wir die gebräuchlichsten Sprachen beurteilen.

(Matthias Rosin/ev)

Reifeprüfung in Pascal

Pascal gilt als hervorragend geeignete Computersprache für Lehre und Unterricht. Zwei Pascal-Systeme treten an, dem Atari ST auch den Schulbereich zu erschließen. Ist der ST reif fürs Abitur?

Computer sind von Natur aus vollkommen respektlos. Unbekümmert dringen sie in alle nur denkbaren Bereiche ein und zeigen selbst davor keine Scheu, in die geheiligten Gefilde des deutschen Schulwesens einzubrechen. Jedem Hüter humanistischen Bildungsgutes muß es wahrhaftig monochrom vor Augen werden, wenn sich seine ihm anvertrauten Zöglinge plötzlich mit einer seltsamen Sprache namens Pascal beschäftigen, statt wie gewohnt Latein oder Griechisch zu büffeln. Und das geschieht dann auch noch an diesen äußerst suspekten Computern.

Sprachbegabt

Doch letztlich muß auch eine so altehrwürdige Institution wie die Schule der modernen Zeit ihren Tribut zollen. Der Informatikunterricht und die Computersprache Pascal haben inzwischen einen festen Platz im Lehrplan erhalten. Die strenge Sprachstruktur und die Problemorientierung machen Pascal ohne jeden Zweifel zu einer idealen Sprache für diesen Anwendungsbereich. Dabei ist Pascal weit entfernt davon, nur eine akademische Lehrsprache zu sein. In Pascal kann man nämlich »richtig« programmieren und Programme erzeugen, die man verkaufen kann. Wer in der Schule Pascal lernt, hat also nicht nur für die Schule, sondern tatsächlich etwas für das Leben gelernt.

Der Atari ST hat sich in der kurzen Zeit seit seiner Markteinführung als ausgesprochen sprachbegabt erwiesen. Neben Logo, C, Basic (hier stottert er noch ein klein wenig), Forth, Modula und verschiedenen anderen Sprachen, spricht und versteht er seit einiger Zeit auch Pascal.

Auf dem Markt gibt es im Moment zwei Pascal-Systeme zu kaufen, das von Atari selbst vertriebene ST-Pascal (Preis: zirka 250 Mark) und das MCC-Pascal (Preis: 340 Mark) der englischen Firma Metacomco. Beide Pascal-Versionen stellen komplette Entwicklungssysteme dar, die auch die Programmierung der grafischen Bedieneroberfläche GEM zulassen. Allerdings sind beide Systeme TOS-Applikationen, also nicht in eine grafische Oberfläche eingebunden.

Zum Lieferumfang des ST-Pascal gehört neben einem 74-seitigen deutschsprachigen Handbuch in haltbarer Ringheftung eine 3½-Zoll-Diskette mit 23 Dateien. Der Besitzer eines Entwicklungspaketes Atari 520 ST findet hier einige alte Bekannte vor, nämlich die GEM-Libraries AESBIND und VDIBIND sowie die Hilfsprogramme BATCH.TTP, RM.PRG und WAIT.PRG aus dem C-Compiler-System von Digital Research. Der Pascal-Compiler besteht aus einer gut 128 KByte großen Datei PASCAL.PRG. Dazu gehören die speziellen Pascal-Libraries PASLIB und PASGRA.O, und eine Textdatei ERRORS.TXT mit den Standard-Fehlermeldungen des Pascal. Vervollständigt wird das Angebot durch einige Steuerdateien zum Compilieren und Linken (die .BAT-Dateien), durch den Linker FASTLINK.PRG, eine Include-Datei SCREEN.INC mit den VT52-Steuercodes, einige Beispielprogramme als Pascal-Quelltext und durch den TOS-Editor EDIT.TTP.

Der Editor ist ein Full-Screen-Editor mit Steuerung durch Cursortasten und Control-Codes. Die Funktionstasten sind nicht belegt. Ebenfalls vorhanden sind einige höhere Editor-Funktionen wie Suchen/Ersetzen und Blockverschieben. Nach Eingabe von CTRL-K können Befehle zur Steuerung des Datenverkehrs mit Diskettenlaufwerken eingegeben werden.

Dieser Editor erlaubt auch das Bearbeiten umfangreicher Quelltexte. Wer allerdings schon einmal einen Text-Editor mit grafischer Bedieneroberfläche benutzt hat, wird den GEM-Komfort schmerzlich vermissen. Da der Compiler des ST-Pascal ASCII-Dateien als Quelltexte erwartet, kann jedoch selbstverständlich jeder Editor benutzt werden, der solche Dateien erzeugt.

Im Gegensatz zu dem nicht allzu komfortablen Editor ist der Compiler PASCAL.PRG geradezu luxuriös ausgestattet. Er kann sowohl durch Anweisungen aus einer Kommandozeile beim Compilerstart, als auch durch Anweisungen aus dem Quelltext in vielfältiger Weise gesteuert werden.

Das Handbuch gibt ausführlich darüber Auskunft. Der Compiler erzeugt eine Objektcode-Datei im 68000-Maschinencode, die ein Linker (FASTLINK.PRG) mit Teilen der Library-Dateien zu fertigen Programmen verbindet. Diese Programme enthalten ein sogenanntes Runtime-Modul und sind deshalb wie andere TOS- oder GEM-Applikationen aus dem GEM-Desktop abrufbar. Der Sprachumfang nach dem ISO-Standard ist beträchtlich erweitert, zum Beispiel durch Implementierung des Variablentyp STRING. Weiterhin sind die sehr schnellen Grafik-Routinen des ST-Betriebssystems, das sogenannte LINE_A-Interface, direkt als Standard-Prozeduren und -Funktionen aufrufbar. Ebenso einfach erfolgt der Aufruf von Betriebssystem- und GEM-Routinen. Spezielle Externdeklarationen (GEMDOS, XBIOS, BIOS und C) im Quelltext bewirken die Einbindung der entsprechenden Library-Module beim Linken. Alle vordefinierten Funktionen erläutert das Handbuch recht ausführlich, sogar mit kleinen Beispielprogrammen. Dennoch kann das Handbuch ein Pascal-Lehrbuch für Anfänger nicht ersetzen. Hierzu sei auf die einschlägige Fachliteratur verwiesen.

Der schon erwähnte Linker FASTLINK.PRG ersetzt die beiden Programme LINK68.PRG und RELMOD.PRG, die mit den ersten Versionen des ST-PASCAL geliefert wurden. Die hohe Geschwindigkeit des Linkvorganges mit FASTLINK.PRG ist in Bild 1 dokumentiert. Besitzer des C-Compilers von Digital Research aus dem Entwicklungspaket können den neuen Blitzlinker auch für das Linken Ihrer C-Programme verwenden.

Schnell wie der Blitz

Bei allem Positiven, was bisher über das ST-Pascal zu vermerken war, darf ein Ärgernis nicht unerwähnt bleiben. Der Pascal-Compiler ist mit einem einfachen Kopierschutz versehen, der aufgrund seiner Primitivität kaum Schutz vor illegalem Kopieren bietet. Dafür bringt er aber beim Arbeiten mit einer RAM-Disk oder, wenn einmal verfügbar, mit einem Festplatten-Speicher unnötigen Zeitverlust beim Compilieren. Selbst wenn alle Dateien der Pascal-Systemdiskette auf eine RAM-Disk oder eine Festplatte übertragen sind, muß sich bei der Compilierung die Pascal-Diskette im Diskettenlaufwerk A befinden. Bei jedem Compilerlauf, egal auf welchem Externspeicher sich der Compiler befindet, wird mehrfach (!) auf

eine besonders präparierte Schutzspur auf der Pascal-Diskette zugegriffen. Dankenswerterweise hatte der Programmierer beim Linker ein Einsehen und verzichtete auf derartige Kindereien.

Das MCC-Pascal wird auf zwei Disketten geliefert, die insgesamt 57 Dateien enthalten. Das mitgelieferte Handbuch mit etwa 200 Seiten ist leider nur in englischer Sprache erhältlich. Wer Englisch gut beherrscht, findet hier allerdings sehr ausführliche Erläuterungen zu Befehlsumfang und Systembedienung. Das System setzt sich aus ähnlichen Elementen zusammen wie beim ST-Pascal. Man findet Editor, Compiler, Include-Dateien zum Aufrufen von GEM-Routinen, Linker und Libraries, sogar gut dokumentierte Assembler-Quelltexte der GEM-Libraries sind vorhanden. Auch hier trifft man zwei wohlbekannte Programme. Wie bei allen Metacomco-Computersprachen wird der TOS-Editor ED.TTP mitgeliefert. Dieser Editor von beträchlicher Leistungsstärke war bereits Bestandteil der zweiten Serie der Atari-Entwicklungspakete, und dürfte entsprechend weit verbreitet sein. Leider hat sich Metacomco noch nicht zur Einbindung in eine GEM-Oberfläche entschließen können.

Als Linker findet eine Updateversion des GST-Linkers aus dem GST-Assembler und dem GST-C Verwendung. Er kann hier aber nicht wie etwa im Assembler aus einer grafischen Bedieneroberfläche gestartet werden. Besitzer des GST-Assemblers können jedoch ohne große Schwierigkeiten das gesamte Pascal-System in die Oberfläche des Assemblers integrieren. Den Pascal-Compiler startet nämlich auch die Option »Run Program« im File-Menü. Der Linker läßt sich wie im Assembler aus dem entsprechenden Link-Menü steuern und starten. Auch die vielfältigen Funktionen zur Dokumentation des Linkvorganges sind ansprechbar.

Der Compiler PASCAL.TTP besitzt viele Optionen, die jedoch nur aus einer Kommandozeile bei Compilerstart, nicht aber aus dem Quelltext, aufrufbar sind. Er erzeugt aus dem Pascal-Quelltext eine Objektcode-Datei im GST-Format mit dem Dateityp .BIN, die durch den GST-Linker bearbeitet werden kann. Es ist aber auch eine Compiler-Option einstellbar, die die Compilierung des Quelltextes in das Format des ASS68 aus dem Entwicklungspaket ermöglicht. Aus diesem Grunde sind die Pascal-Library PASLIB, die GEM-Library GEMLIB und das Runtime-Modul STARTUP als .BIN-Dateien für den GST-Linker und als .O-Dateien für Digital Research Linker (LINK68 oder FASTLINK) auf den Disketten vorhanden.

Zwei Disketten voll Pascal

Zum Test und zum Vergleich mit dem ST-Pascal wurde nur die GST-Option verwendet. Die fertigen Programme sind wie beim ST-Pascal auch ohne Pascalsystem lauffähig und wie normale GEM- oder TOS-Applikationen zu benutzen.

Was leistet der MCC-Pascal-Compiler? Um es gleich vorweg zu sagen, MCC-Pascal ist nur etwas für ausgesprochene Pascal-Puristen. Der Sprachumfang umfaßt genau das ISO-Standardpascal, nicht mehr, aber auch nicht weniger. Der Variablentyp STRING fehlt ebenso wie die Funktion KEYPRESS. Dinge also, die die Pascalprogrammierung erleichtern. Leider kann die vorliegende Version des MCC-Pascal keine Betriebssystem- oder LINE_A-Aufrufe durchführen. Die GEM-Funktionen sind über eine Gruppe von INCLUDE-Dateien zugänglich, die EXTERN-Deklarationen von AES- und VDI-Routinen sind entsprechend den Digital Research-Spezifikationen zum GEM-System enthalten. Die eigentlichen Routinen befinden sich in der Library GEMLIB, die allerdings noch nicht ganz fehlerfrei zu arbeiten scheint.

Bei einem Vergleich der beiden Pascal-Versionen wurden die Zeiten für Compilierung und Linken des Benchmark-Klassikers »Sieb des Eratosthenes« sowie die Laufzeiten der erzeugten Programme für einen Durchlauf zur Ermittlung der Primzahlen gemessen (Ergebnisse in Bild 1).

Der MCC-Compiler arbeitet sehr schnell. Er ist beim Arbeiten mit der Diskette schneller als der ST-Pascal-Compiler mit einer RAM-Disk. Genau umgekehrt verhält es sich hinsichtlich der Linkdauer. Hier hat FASTLINK eindeutig die Nase vorn. Geradezu sensationell ist jedoch der Unterschied in der Laufzeit der Programme. Mit gut 17 Sekunden braucht das mit MCC-PASCAL erzeugte Programm länger als das entsprechende Turbo-Pascal-Programm unter CP/M-Emulation.

Bild 2 gibt einen weiteren Test wieder, der neben der Laufzeit der Programme die Rechengenauigkeit mit Fließkommazahlen überprüft. Dabei wird in einer WHILE...DO-Schleife von einer vorgegebenen Zahl solange der Wert 0,1 subtrahiert, bis die vorgegebene Zahl den Wert 0 unterschreitet. Dies müßte bei einem Startwert 100000 nach genau einer Million Substraktionen erreicht sein. Wie Bild 2 zeigt, besteht der ST-Pascal-Compiler diesen Tast mit Bravour. Erst bei Startwert 100000 treten merkbare Ungenauigkeiten auf. Der MCC-Pascal-Compiler macht bereits bei Startwert 100 sichtbare Fehler, bei Startwert 100000 wird der Wert 0 bereits nach 990564 Substraktionen erreicht.

Aufgrund dieser Ergebnisse fällt eine abschließende Beurteilung leicht. Das MCC-Pascal ist teurer, hat den geringeren Sprachumfang, erzeugt langsamere Programme und rechnet ungenauer. Auf der Plusseite lassen sich die geringere Compilierzeit und der bessere Editor verbuchen.

In allen wichtigen Punkten ist jedoch der ST-Pascal-Compiler die eindeutig bessere Wahl. Trotz des geringeren Preises besitzt er neben dem kompletten Sprachumfang des ISO-Standard-Pascal viele wichtige Erweiterungen, die die Fähigkeiten des Atari ST besser ausnutzen. Auch die Laufgeschwindigkeit der erzeugten Programme läßt kaum Wünsche offen. Die Abiturreife kann ohne Bedenken zuerkannt werden. (W. Fastenrath/hb)

Compilierzeiten »Sieb des Eratosthenes«

	Diskettenbetrieb		RAM-Disk		Laufzeit
	Compiler	Linker	Compiler	Linker	
MCC-PASCAL	12,7 sec	106,0 sec	1,9 sec	9,8 sec	17,3 sec
ST-PASCAL	51,0 sec	29,7 sec	14,8 sec	1,1 sec	0,7 sec

Bild 1. S wie super, T wie Tempo: Supertempo von ST-PASCAL

Rechengenauigkeit mit REAL-Zahlen

	MCC-PASCAL			ST-PASCAL		
Startwert	Endwert	Subtraktionen	Zeit	Endwert	Subtraktionen	Zeit
100	0,0010	1000	1,3	0,0000	1000	0,5
1000	0,0971	10000	7,6	0,0000	10000	3,0
10000	0,0491	100015	72,8	0,0000	100000	25,8
100000	0,0964	990564	660,6	0,0853	1000000	241,0
200000	-	-	-	0,0437	2000000	478,3

Bild 2. Rechnen will gelernt sein

Pascal auf dem C64

Pascal ist an Universitäten und in Software-Häusern gleichermaßen anerkannt. Probleme vollständig zu analysieren und strukturiert zu programmieren - das bringt Pascal Ihnen bei.

Die von Wirth entwickelte Sprache Pascal lief zunächst nur auf Großrechnern. Seinen Siegeszug auf Mikrocomputern trat Pascal an, als auf der Universität von California in San Diego (UCSD) das berühmte UCSD-Pascal entstand. Es ist eine Anpassung des Wirth'schen Standard-Pascal an einen Mikrocomputer und enthält sowohl Einschränkungen als auch Erweiterungen.

Pascal-Versionen für den Commodore 64 gibt es etwa seit 1983. Dabei handelt es sich in der Regel um »Enkel« des Wirth'schen Urcompilers, bei denen allen man gewisse Abstriche machen muß: Sie erzeugen keine Maschinensprache für den C64, sondern einen Zwischencode, auch P-Code genannt.

Der Compiler übersetzt in P-Code

Im Gegensatz zur Standard-Sprache Basic benutzt UCSD-Pascal einen Compiler, der den Pascal-Quellcode in P-Code übersetzt. Der P-Code (P steht für Pseudo) ist einer wirklichen Maschinensprache sehr ähnlich. Man kann sich diese Sprache als Modell eines Computers vorstellen. Es besitzt eigene Register, einen Stack, einen Heap für dynamische Variablen und einen Prozessor, der den P-Code ausführt. Da all dies die Software zuwege bringt, die Maschine also nur virtuell vorhanden ist, läuft P-Code um einiges langsamer als echter Maschinencode. Die Laufzeiten liegen dann um etwa zwei- bis fünfmal höher als bei echtem Maschinencode.

Man sollte aber die Vorteile dieses Verfahrens nicht übersehen. P-Code schlägt Basic in der Geschwindigkeit immer noch bei weitem (etwa 10 mal so schnell). Der erzeugte P-Code ist kompakter als der übliche Maschinencode und benötigt daher weniger Speicherplatz.

Gegenüber Standard-Pascal weist KMMM-Pascal einige Funktionen mehr auf. Dazu gehört vor allem eine komplette String-Behandlung. Auch Bit-Operationen auf Integer-Variablen sind erlaubt. Maschinensprache-Routinen können von KMMM aus aufgerufen werden. Peek und Poke sind ähnlich wie in Basic zu verwenden.

Das Programm besteht aus einem Editor, einem Compiler und einem Translator. Der Editor ist ein formatfreier Full-Screen-Editor mit überdurchschnittlichem Komfort (im Vergleich zum Basic-Editor). Er besitzt eine Funktion zur Überprüfung der Syntax und belegt alle Funktionstasten. Eine häufig benötigte Befehlssequenz kann als Makro definiert werden.

Der KMMM-Editor erzeugt eine sequentielle Datei, weshalb auch andere Editoren den Programmtext bearbeiten können. Danach wird der Compiler geladen. Er erzeugt bei Bedarf ein Listing auf dem Drucker und wandelt das Quellprogramm in P-Code um. Nach dem Laden des Translators übersetzt dieser nun den P-Code in schnelle Maschinensprache.

Oxford-Pascal unterstützt weitgehend alle Elemente von Standard-Pascal. Eine spezielle String-Verarbeitung unterblieb. Es verwendet statt dessen Variablen vom Typ PACKED ARRAY OF CHAR. Dafür stehen zusätzliche Befehle zum Erzeugen von Grafik und Sound bereit. Die Möglichkeiten des C64 lassen sich also auch von Pascal aus nutzen. Der Bildschirm kann sogar in einen Grafik- und in einen Textbereich aufgeteilt werden. Solche Programme laufen allerdings deutlich langsamer.

Unterstützt werden auch Peek und Poke, das Einfügen von Routinen in Maschinensprache und die Programmierung der eingebauten Uhr. Ärgerlich ist lediglich die zu dünn geratene Dokumentation von etwa 50 Seiten, bei der auch noch die Nummerierung der Seiten und ein Stichwortverzeichnis fehlen.

Die Entwicklung von kleineren Programmen gestaltet sich mit Oxford-Pascal angenehm. Der Basic-Editor ist lediglich etwas erweitert. Ansonsten arbeitet man genauso wie mit einem Basic-Programm. Der Compiler und das übersetzte Programm befinden sich gleichzeitig im Speicher. Allerdings fehlen in diesem Modus einige Befehle. Erst im Disk-Modus verfügt man über den gesamten Sprachumfang, muß aber dann die Zeit für das Laden des Compilers und das Speichern der Programme in Kauf nehmen.

Oxford-Pascal übersetzt in P-Code. Es gibt aber einen Locate-Befehl, der eine P-Code-Datei in eine von Basic aus ladbare Datei umwandelt. Eine solche Datei benötigt dann das Oxford-System nicht mehr.

Profi-Pascal hat gegenüber den obigen Pascal-Versionen einen entscheidenden Vorteil. Der Zugriff auf die 1541-Floppy geht wegen der zusätzlichen Diskettenroutinen dreimal so schnell vor sich. Allerdings wurde dieser Vorteil auch mit einem entscheidenden Nachteil erkauft. Profi-Pascal besitzt nämlich praktisch ein eigenes Betriebssystem mit Editor, Compiler und Hilfsprogrammen. Auf die Kompatibilität zu anderen Programmen und Dateien wird damit verzichtet.

Nach dem Laden erscheint ein Hauptmenü, von dem die anderen Programmteile (Editor, Compiler) aus geladen werden. Profi-Pascal enthält viele zusätzliche Routinen über den normalen Sprachumfang hinaus. So ist beispielsweise der Zugriff auf den gesamten Arbeitsspeicher auf sehr elegante Weise möglich. Der in Standard-Pascal nicht vorgesehene Typ String erlaubt die Manipulation von Zeichenketten. Der Direktzugriff auf Sätze innerhalb einer Datei ist mit der zusätzlichen Prozedur Seek möglich.

Als weitere Eigenschaften sind das Verketten von Dateien und das Segmentieren von Programmen zu nennen. Programme mit Segment-Prozeduren laden einen Teil des Codes während des Programmlaufs in den Arbeitsspeicher nach.

Profi-Pascal besitzt komfortable Möglichkeiten, Assembler-Code in das Programm einzubinden. Schließlich enthält es einen eigenen Assembler. Der Pascal-Code selbst wird allerdings nicht in Maschinensprache, sondern in P-Code übertragen. Nachteilig ist, daß der Compiler nur auf der (natürlich kopiergeschützten) Original-Diskette arbeiten kann.

Pascal für Einsteiger und Profis

Eine Pascal-Version für den preiswerten Einstieg in die interessante Sprache ist ganz neu im Markt&Technik Verlag erschienen. Es handelt sich dabei um einen kompletten Pascal-Kurs in Buchform, der dazu sehr *schnell* und leistungsfähig ist und ohne Aufpreis dem Buch gleich auf Diskette beiliegt. Neben dem Compiler enthält die Diskette noch viele nützliche Beispielprogramme. Dieses »Pascal für den C 64« nutzt für seine Programme den gesam-

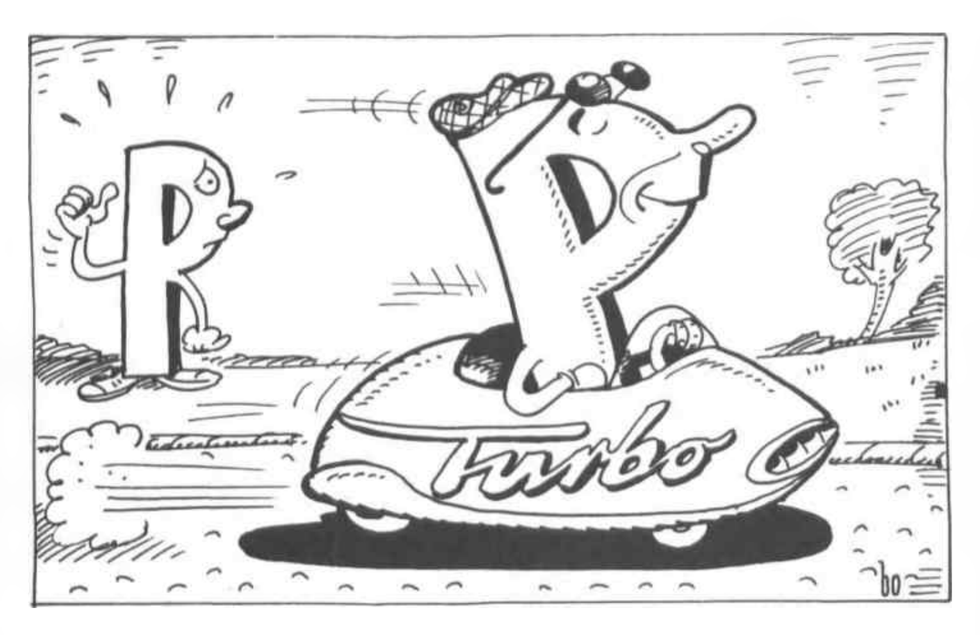


ten Speicher des C 64 aus. Ein Full-Screen-Editor ermöglicht eine komfortable Programmeingabe. Der Compiler akzeptiert den gesamten Standard-Sprachumfang mit einigen Erweiterungen. Das Buch enthält einen Einführungskurs mit vielen Beispielen und Übungsaufgaben für den Anfänger sowie Tips und Tricks für den Profi. Mit gutem Gewissen kann man Buch und Compiler als preiswerte Alternative zu den bisher bekannten Pascal-Versionen für den C64 empfehlen. Denn bei diesem Pascal-Compiler gibt es trotz des extrem niedrigen Preises keinerlei Abstriche an Sprachumfang oder Dokumentation. Als Autor von Buch und Compiler zeichnet Florian Matthes, den wir auch als Autoren für den Pascal-Kurs in diesem Sonderheft gewinnen konnten.

Die Entscheidung für einen der vier Compiler ist sicherlich nicht einfach. Wer einfache Bedienung wünscht, dem sei Oxford-Pascal empfohlen. Reine Maschinensprache wird nur von KMMM-Pascal erzeugt. Profi-Pascal wiederum bietet den größten Sprachumfang.

Und ein C 64-Fan, ob Einsteiger oder Profi, wird schließlich mit Markt&Technik-Pascal am meisten Spaß haben.

(Anton Gruber/cg)



Turbo-Pascal: der Renner

Seit etwas mehr als zwei Jahren gibt es einen Pascal-Compiler, der auf dem Computermarkt Furore gemacht hat: Turbo-Pascal. Was das Programm alles leistet, ist schon fast unheimlich.

Die Gründe für den grandiosen Erfolg von Turbo-Pascal sind recht einfach nachzuvollziehen. Da ist zuerst einmal der (damals) konkurrenzlos niedrige Preis. In den USA kostet Turbo-Pascal etwa 50 Dollar – in Deutschland um die 200 Mark. Ein Direktimport lohnt sich aber dennoch nicht, weil Sie Zoll und Verpackungsgebühren bezahlen müßten, die den scheinbaren Preisunterschied wieder ausgleichen.

Einen Pascal-Compiler für 200 Mark – das gab es vor Turbo-Pascal nirgends auf dem Computermarkt. So fanden sich nicht wenige, denen das Risiko eines Fehlkaufs klein genug erschien und sich aus diesem Grund Turbo-Pascal gekauft haben. MS-Pascal von Microsoft und Pascal/MT+ von Digital Research, die beiden vormals hauptsächlich benutzten Compiler, kosteten je nach Betriebssystem zwischen 1000 und 2000 Mark. Für MS-Pascal gilt dies heute noch, während Pascal/MT+ in einer Version für den Schneider CPC 6128 inzwischen auch auf ungefähr 200 Mark herunterging. Damit zielt es direkt auf den Turbo-Pascal-Markt.

Allein der Preis kann aber dennoch nicht für den Erfolg von Turbo-Pascal verantwortlich sein. Schließlich gibt es heute auch schon Pascal-Compiler ab 100 Mark (zum Beispiel Nevada-Pascal), die trotz ihres günstigen Preises eher ein Schattendasein führen.

Zählen wir also weiter Turbo-Pascal-Vorzüge auf: Turbo-Pascal ist ein integriertes Programmpaket, das aus einem Editor, dem eigentlichen Compiler und einer Laufzeitumgebung besteht. Was das heißt, kann nur jemand ermessen, der schon mit anderen Compilersprachen gearbeitet hat. Ein typischer Arbeitszyklus sieht dort etwa so aus: Texteditor laden (meistens wird keiner mitgeliefert, also muß Wordstar oder ein anderes normales Textprogramm dafür herhalten), Datei anlegen, Text eintippen, Datei speichern, Editor beenden und Compiler aufrufen. Dieser übersetzt das Programm in Maschinen-Quellcode oder einen Zwischencode, zum Beispiel den p-Code von UCSD-Pascal.

Hier scheiden sich dann die (Compiler-)Geister: Manche benötigen einen Assembler, der Maschinencode erzeugt, andere einen Linker, der die Programm-Module zusammenkettet und wieder andere einen Laufzeit-Interpreter, der die Programme – ähnlich einem Basic-Interpreter – in der Zwischensprache ausführt. Wenn der Compiler allerdings einen Fehler meldet, muß der Programmierer wieder mit dem Editor heran und die Fehler ausbessern. Stürzt das Programm gar völlig ab, geht die Fehlersuche ganz von vorne los.

Ganz anders dagegen arbeitet Turbo-Pascal. Der Programmierer gelangt in eine Art »Benutzerebene«, von der aus er Befehle geben kann. So ruft er mit »E« für »Edit« den eingebauten Texteditor auf. Dieser versteht – Segen für alle Wordstar-Freaks – eine Untermenge der Wordstar-Kommandos. Er ist aber auf das Editieren von Pascal-Programmen hin erweitert worden. So gibt es eine »Auto-Indent«-Funktion. Diese bewirkt, daß der Cursor nach Drücken der Enter-Taste nicht an den Anfang der nächsten Bildschirmzeile, sondern unter den Beginn der letzten Programmzeile gesetzt wird. Damit unterstützt Turbo-Pascal wirkungsvoll das Prinzip der Texteinrückungen, die Pascal-Programme so gut lesbar machen. Auch ist der Turbo-Editor ein ganzes Stück schneller als Wordstar. Beispielsweise führt er den Bildschirmaufbau und die Invertierung von Textblöcken mit Höchstgeschwindigkeit durch – auch eine Rechtfertigung für »Turbo« im Namen »Turbo-Pascal«. Dieser Geschwindigkeitsvorteil liegt allerdings zum Teil daran, daß der gesamte bearbeitete Programmtext im Speicher liegen muß, während Wordstar auch Diskettentexte bearbeiten kann. Trotz dieser Einschränkung eignet sich der

Turbo-Editor nebenbei hervorragend zur Textverarbeitung. So sind die Blockkopierbefehle sogar auf dem CPC 464 und CPC 664 von Schneider ohne Speichererweiterung funktionsfähig.

Sobald der Benutzer den Pascal-Quellcode fertig eingegeben hat, drückt er in Wordstar-Manier CTRL-KD und gelangt wieder in die Kommandoebene. Dort kann er das Diskettenlaufwerk wechseln, neue Disketten anmelden, die Datei speichern und andere Dateien laden. »D« gibt ein Directory der angemeldeten Diskette aus und hat gegenüber dem CP/M-Kommando DIR den Vorteil, daß der freie Disketten-Speicherplatz mit angezeigt wird. Der Benutzer erfährt aus dem Hauptmenü auch, wieviel Speicherplatz der Quelltext belegt und wieviel Platz noch frei bleibt. Mit »C« läßt sich der Compiler starten, der das Programm mit extrem hoher Geschwindigkeit (»Turbo«) übersetzt. Er besitzt jede Menge Fehlermeldungen, die meist sehr informativ sind. Sobald er einen Fehler entdeckt, gibt Turbo-Pascal die (meistens) passende Meldung aus und kehrt direkt in den Turbo-Editor zurück. Der Cursor steht dann an der Stelle, an der der Fehler auftrat. Wenn der Speicherplatz knapp wird, können Sie aber auch die Fehlermeldungen aus dem Programm herauswerfen und haben damit etwa 2 KByte mehr Speicherplatz zur Verfügung. Alle Fehler erscheinen dann nur noch mit ihren Nummern, die im – gut gemachten – Handbuch nachzuschlagen sind.

Müssen Sie sich mit fehlendem Speicherplatz herumschlagen (auf dem CPC 464 und CPC 664 verbleiben nur noch etwa 6 beziehungsweise 8 KByte), bleiben aber auch noch andere Wege, große Programme zu übersetzen. Da wären zuerst einmal die Include-Files. Das sind Dateien, die separat auf der Diskette stehen und erst bei der Übersetzung in den Programmcode integriert werden. Außerdem läßt Ihnen Turbo-Pascal die Wahl, ob der Objektcode des Programms im Speicher abgelegt werden soll (Compile To Memory) oder auf Diskette. Bei letzterem können Sie sich zwischen dem Dateityp »COM« und »CHN« entscheiden. Der Unterschied besteht darin, daß im ersten Fall das erzeugte Programm mit einer Laufzeitbibliothek ausgestattet ist. Diese kostet zwar einiges an Speicher- und Diskettenplatz, erzeugt aber Programme, die unabhängig von Turbo-Pascal arbeiten.

Die Programme sind – wie bereits gesagt – reiner Maschinencode. Die 8-Bit-Variante von Turbo-Pascal erzeugt Z80-Code und ist damit eines der wenigen CP/M-Programme, die nicht auf Computern mit 8080/8085-Prozessor gestartet werden können, sondern den Z80-Chip voraussetzen. Andererseits sind die Z80-Erweiterungen sehr leistungsfähig und bewirken kompakteren und schnelleren Maschinencode.

Was den Programmierer aber wohl am brennendsten interessiert, ist der Befehlsvorrat des Compilers – ein weiterer Pluspunkt für Turbo-Pascal. Während viele Pascal-Compiler der niedrigeren Preisklassen »Subsets« des Sprachstandards sind, also nur Teile der Sprache verstehen, ist Turbo-Pascal ein »Superset« – und was für eines!

Niklaus Wirth entwickelte Pascal eigentlich nur als Lehr- und Unterrichtssprache. Dementsprechend fehlen in Standard-Pascal viele nützliche und dringend notwendige Dinge. Turbo-Pascal wartet mit einer ganzen Menge von diesen auf. Da gibt es zum Beispiel den Datentyp STRING, den man nun nicht mehr durch einen ARRAY OF CHAR simulieren muß. Stilgerecht fehlen auch nicht die entsprechenden Funktionen und Prozeduren zur Stringbehandlung. Selbstverständlich wird die Dateibehandlung auf Diskette und Festplatte samt Relativ- und Direktzugriffdateien unterstützt. Neu ist auch der Datentyp BYTE.

Für Kenner ein Plus: Maschinennähe

Für Maschinensprache-Kenner, denen die Entwicklung von reinen Maschinencode-Programmen zu mühsam ist, hält Turbo-Pascal eine Reihe von maschinennahen Erweiterungen bereit: SHR (Shift Right) und SHL (Shift Left) erlauben das Links- und Rechtsschieben von Bits in Zahlen und Variablen. Die Funktionen HI und LO liefern das High- und Lowbyte einer Integervariablen in eine Byte-Variable:

```
Program HiLow;
Var i:Integer;
Begin
 ReadLn(i);
 WriteLn(Hi(i),' ',Lo(i));
End.
```

Dieses kleine Beispielprogramm liest eine 16-Bit-Zahl von der Tastatur ein und gibt deren High- und Lowbyte aus.

Weitere maschinennahe Funktionen sind ADDR zur Ermittlung der Adresse einer Variablen und PORT zum direkten Zugriff auf die Prozessor-Ports. Das Pseudo-Array MEM (»Memory«) simuliert die beliebten PEEK- und POKE-Befehle aus Basic. Diese Sprache erscheint im Gegensatz zu Turbo-Pascal regelrecht archaisch.

Auf einer viel höheren Ebene arbeiten die Befehle zur Verwaltung dynamischer Variablen, nämlich MARK, RELEASE und DISPOSE.

Natürlich präsentiert sich Turbo-Pascal nicht »nackt«. Neben dem TURBO.COM-File, das mit 31 KByte Länge das Hauptprogramm darstellt, sowie TURBO.MSG (System-Meldungen) und TURBO.OVR (ein 2KByte-Overlay) findet sich noch MC.PAS, das als Demonstration der Leistungsfähigkeit des Compilers dient. MC.PAS ist ein kleines Tabellenkalkulationsprogramm, genannt MicroCalc. Es ist verständlicherweise nicht so leistungsfähig wie seine großen Brüder (etwa Multiplan, Visicalc oder Supercalc), aber es erfüllt seinen Zweck recht gut. Es wurde allerdings auch gar nicht zur »richtigen« Arbeit geschrieben, sondern soll vielmehr exemplarisch die Fähigkeiten von Turbo-Pascal zeigen. Wenn die Kalkulationsaufgaben nicht allzu kompliziert werden, erspart es trotzdem das Geld für die Anschaffung eines anderen Programms.

Zum Lieferumfang gehört ferner ein Programm, das andere Pascal-Programme auf dem Drucker auflistet. LISTER.PAS erzeugt an den passenden Stellen einen Seitenvorschub und kann auf Wunsch Zeilennummern in den Text einfügen und Pascal-Schlüsselwörter unterstreichen.

Für die Schneider-Computer gibt es – gegen Aufpreis, versteht sich – eine Grafikerweiterung, die die hervorragenden Fähigkeiten dieser Computer auf diesem Gebiet unterstützt. Unter anderem dient TURTLE.PAS als einfaches Zeichenprogramm nach dem Prinzip der Logo-Schildkröten-Grafiken. WINDOW.PAS zeigt, wie das »Fensterln« unter Pascal funktioniert.

Immer mehr Firmen versuchen, sich an den Erfolg von Turbo-Pascal anzuhängen und liefern Programmsammlungen für alle möglichen Anwendungszwecke. Zum Beispiel werden da Turbo-Lader, Turbo-Machine, Turbo-Screen oder Turbo-Data angeboten. Von Heimsoeth-Software, der deutschen Vertriebsfirma von Turbo-Pascal, stammen unter anderem Turbo-Graphix, Turbo-Editor und Turbo-Gameworks. Doch aufgepaßt: Turbo-Pascal gibt es auch für IBM-Computer unter MS-DOS. Viele der Programmpakete arbeiten nur unter diesem Betriebssystem und sind für CP/M-Benutzer unbrauchbar. Das betrifft zum Beispiel auch einige der gerade genannten Programme.

Leider vernachlässigt Borland zur Zeit etwas die CP/M-80-Version von Turbo-Pascal. So hat die 8-Bit-Variante vom »Lifting« in der neuen 3.0-Version kaum profitiert, während das MS-DOS-Pendant noch weiter verbessert wurde. Dennoch: An Turbo-Pascal gibt es nicht mehr allzu viel besser zu machen. Es ist einfach schon nahezu perfekt.

(Martin Kotulla/hg)

Die C-Crew im Test



C-Compiler sind in aller Munde, und die Beliebtheit der Sprache C steigt ständig. Steigen auch Sie ein in diese Supersprache. Wir zeigen Ihnen den richtigen C-Compiler für den Atari ST.

Das Angebot an C-Compilern für den Atari ST wächst: Kein Wunder, denn C ist eine Sprache, die das Betriebssystem des Atari ST voll unterstützt. Das Betriebssystem wurde auch in dieser Sprache verfaßt.

Da Softwarehäuser C-Programme wegen ihrer Portabilität sehr schätzen, ist im Lieferumfang des Entwicklungspakets zum Atari ST auch der C-Compiler von Digital Research enthalten.

Dieser Compiler, als Teil eines Profi-Systems, hat zwar noch einige Fehler, aber es läßt sich vernünftig damit arbeiten. Den Heimanwender interessiert das System wegen des hohen Preises wohl weniger. Atari gibt den Compiler leider nur mit dem Entwicklungspaket ab - einzeln ist er nicht zu beziehen.

Es gibt aber inzwischen gute Alternativen. Allerdings ist wegen des begrenzten Angebots die Auswahl noch nicht sehr groß; doch einige Firmen kündigten bereits eigene Compiler an, und allzulange werden diese wohl auch nicht mehr auf sich warten lassen. Für kommende rosige Zeiten unterbreiten wir Ihnen hier einige Auswahlkriterien für C-Compiler.

Als erstes sollten Sie darauf achten, wie vollständig die Sprache implementiert ist. Mit allen 28 Schlüsselwörtern ist der Sprachumfang vollständig. Viele Entwickler verzichten jedoch auf das eine oder andere Leistungsmerkmal, und da muß man sich genau überlegen, inwieweit das für die eigenen Zwecke tragbar ist.

Erfahrungsgemäß wächst mit dem Wissen auch der Anspruch. Schnell stellt man fest, daß der Compiler, der noch vor kurzem genügte, für die nächste Anwendung große Mängel aufweist. Man sollte sich vorher genau über das Produkt informieren und eventuell etwas tiefer in die Tasche greifen, um vor einer Enttäuschung sicher zu sein.

Eine »Einsteigerversion« ist der C-Compiler von GST. Er kann nur ganze Zahlen verarbeiten und führt keine Fließkomma-Arithmetik durch. Damit beschränkt sich sein Einsatzgebiet auf Systemprogrammierung und weitere Anwendungsprogramme, wie Textverarbeitung, Datenbanken und Spiele.

Natürlich kann man sich die Floatingpoint-Routinen auch selbst schreiben, aber es gibt bereits andere C-Compiler für den Atari-ST, die nicht viel mehr kosten und bereits alles beinhalten. Beim GST-Compiler verzichtete man auch auf die Strukturen (das Schlüsselwort »structure« fehlt). Um gute Algorithmen zu schreiben, ist man aber auf Datenstrukturen, und damit auf dieses Schlüsselwort angewiesen.

Kompletter Sprachumfang

Wer etwas tiefer einsteigen möchte in die faszinierende Programmiersprache C, sollte lieber zum Compiler von Lattice greifen. Er ist der einzige, der den kompletten Sprachumfang von Kernighan und Ritchie besitzt. Beim DRI-Compiler arbeiten die Funktionen scanf, für formatierte Ausgabe, und getchar, um ein Zeichen einzulesen, nicht korrekt. Der Lattice-Compiler hat aber wiederum andere kleine Fehler. Beim Arbeiten mit Fenstern macht er Schwierigkeiten, da er eine falsche »Window_handle«-Nummer zurückgibt. Ansonsten darf man ihn getrost als die zur Zeit beste Implementation für den ST bezeichnen.

Eines der wichtigsten Kriterien für einen C-Compiler sind seine Libaryfunktionen.

Zum Standard-C gehört eine Standard-Libary. Informationen über den Umfang erhält man aus der Sprachbeschreibung von Kernighan und Ritchie (siehe Literaturverzeichnis), die auch den C-Standard definierte.

Dieser sogenannte K&R-Standard sollte in der Libary auf keinen Fall fehlen.

Wichtig ist außerdem, wie die Zugriffe auf GEM und das Betriebssystem geregelt sind. Für die GEM-Programmierung selbst bietet der GST-Compiler am meisten an. Bei den Betriebssystemaufrufen macht er leider große Abstriche.

Die Libary enthält keine der Funktionen »gemdos«, »bios« und »xbios«, sondern lediglich eine kleine Auswahl vordefinierter »gemdos«-Funktionen. Für den Profi ist das kein Problem: Mit nur wenigen Assembler-Kenntnissen kann man sich die fehlende Betriebssystem-Schnittstelle selber stricken. Dennoch, auch diese Einschränkung sollten Sie bei einer Kaufentscheidung berücksichtigen.

Hier schneidet wiederum das Lattice-Produkt am besten ab. Seine Libary-Funktionen sind außerordentlich umfangreich.

Zwar nicht ausschlaggebend, aber erwähnenswert, bleiben die Extras, wie zum Beispiel ein Editor. Der GST-Compiler zeichnet sich durch einen sehr guten Editor aus. Entscheidender aber sind Hilfen zur Fehlersuche.

Nun kennen Sie alle zur Beurteilung eines C-Compilers wichtigen Kriterien. Haben Sie alle für Sie wichtigen Punkte beachtet, dann lassen Sie sich von der Leistungsfähigkeit Ihres C-Compilers überraschen. (hb)

Info: C-Compiler von Digital Research Inc. Atari Corp. (Deutschland) GmbH, Frankfurter Str. 89-91, 6096 Raunheim, Tel: 06142/41081. Nur erhältlich mit dem Entwicklungspaket für den Atari ST, Preis 969 Mark.

C-Compiler von GST in Kürze bei Atari erhältlich. Preis zirka 300 Mark.

Lattice-C-Compiler von Metacomco. Vertrieb über Philgerma, Ungererstr. 42, 8000 München 40, Tel. 089/395551, Preis 380 Mark.

Small-C – ein C-Compiler unter CP/M

C wird in Zukunft immer mehr an Bedeutung gewinnen. C verbindet die Eigenschaften einer Hochsprache mit denen einer Maschinensprache. C-Programme sind sehr schnell. Drei gute Gründe für C! Unter CP/M gibt es jetzt einen preiswerten Einstieg.

C ist eine Sprache, die in letzter Zeit an Bedeutung gewinnt. Das berühmte Betriebssystem Unix ist in C geschrieben. Und immer mehr kommerzielle Software wird in C entwickelt. Es gibt einige C-Compiler, die unter dem Betriebssystem CP/M laufen. Stellvertretend für diese präsentieren wir Small-C, ein Entwicklungssystem, das eine umfangreiche Programmbibliothek (Quellcode – ebenfalls in C) mitbringt.

Das Programmpaket Small-C enthält außer dem C-Compiler noch einen Makro-Assembler zur Erzeugung von relokatierbarem 8-Bit-Objektcode für 8080/Z80-Prozessoren und einen Linker zum Binden einzelner Module zu lauffähigen Programmen. Außerdem bereichern noch eine Menge hilfreiche und gleich mitgelieferte Dienstprogramme (Tools) dieses Paket. Als Hardware-Voraussetzung benötigt man das Betriebssystem CP/M und mindestens 56 KByte Hauptspeicher.

Der Commodore 128 im CP/M-Modus findet in diesem Entwicklungssystem ein geeignetes Werkzeug, um mit der Sprache C die ersten Gehversuche zu unternehmen. Aber auch für alte Hasen liefert es eine Menge Anregungen zum Programmieren. Das gesamte Paket ist bis auf ein paar arithmetische und logische Funktionen selbst in C geschrieben.

C in C geschrieben

Der C-Compiler erzeugt aus einem C-Quellcode einen Assembler-Quellcode und erst dieser wird in einen relokatierbaren Objektcode übersetzt. Dazu eignet sich der mitgelieferte Small-MAC-Assembler oder der M80-Assembler von Microsoft. Der Sprachumfang umfaßt lediglich eine Untermenge der Sprachdefinition nach Brian W. Kernighan und Dennis M. Ritchie, den Entwicklern der C-Sprache. Die Definitionen, Fließkomma-Datentypen (float,double), sizeof, mehrdimensionale Arrays, Zeigerarray, Strukturen (struct,union), Bit-Felder und casts fehlen. Daß aber trotzdem größere Programme erzeugt werden können, zeigt das Small-C-Paket selbst. Es sind alle Unix-Funktionen implemetiert, soweit sie in einer fremden Umgebung anwendbar sind.

Der Small-MAC-Assembler beinhaltet ein ganzes Paket von Programmen. Er enthält im einzelnen einen Makro-Assembler (MAC), einen Linker (LNK) und einen Bibliotheksverwalter (LIB). Zusätzlich stehen noch ein Lader (LGO), ein CPU-Anpassungsprogramm (CMIT) und ein Dump-Programm (DREL) zur Verfügung. Der Makro-

Ein-/Ausgabe

fopen(name, mode)
freopen(name, mode, fd)
fclose(fd)
fgetc(fd)
ungetc(c, fd)
getchar()
fgets(str, sz, fd)
fread(ptr, sz, cnt, fd)
read(fd, ptr, cnt)
gets(str)
feof(fd)
ferror(fd)
clearerr(fd)
fputc(c, fd)
putchar(c)
fputs(str, fd)
puts(str)
fwrite(ptr ,sz, cnt, fd)
write(fd, ptr, cnt)
fflush(fd)
cseek(fd, offset, from)
rewind(fd)
ctell(fd)
unlink(name)
rename(old, new)
auxbuf(fd, size)
iscons(fd)
isatty(fd)
printf(str, arg1, ...)
fprintf(fd, str, arg1, ...)
scanf(str, arg1, ...)
fscanf(fd, str, arg1, ...)

Formatkonvertierung

atoi(str)
atoib(str, base)
itoa(nbr, str)
itoab(nbr, str, base)
dtoi(str, nbr)
otoi(str, nbr)
utoi(str, nbr)
xtoi(str, nbr)
itod(nbr, str, sz)
itoo(nbr, str, sz)
itou(nbr, str, sz)
itox(nbr, str, sz)

Zeichenketten

left(str)
pad(str, ch, n)
reverse(str)
strcat(dest, sour)
strncat(dest, sour, n)
strcmp(str1, str2)
lexcmp(str1, str2)
strncmp(str1, str2, n)
strcpy(dest, sour)
strncpy(dest, sour, n)
strlen(str)
strchr(str, c)
strrchr(str, c)

Zeichenklassifizierung

isalnum(c)
isalfa(c)
isascii(c)
iscntrl(c)
isdigit(c)
isgraph(c)
islower(c)
isprint(c)
ispunct(c)
isspace(c)
isupper(c)
isxdigit(c)
lexorder(c1, c2)

Zeichenumwandlung

toascii(c)
tolower(c)
toupper(c)

mathematisch

abs(nbr)
sign(nbr)

Programmkontrolle

calloc(nbr, sz)
malloc(nbr)
avail(abort)
free(addr)
getarg(nbr,str,sz,argc,argv)
poll(pause)
exit(errkode)

Tabelle 1. Vollständiges Verzeichnis der Funktionen von Small-C

CHG(Change)	Ersetzen von Zeichenketten in Textdateien
CNT(Count)	Zählen von Zeichen, Wörtern oder Zeilen
CPY(Copy)	Kopieren von Textdateien
CPT(Crypt)	Ver- und entschlüsseln von Dateien
DBT(Detab)	Ersetzen von Tab-Zeichen durch Leerzeichen
EDT(Edit)	Zeileneditor
ETB	Gegensatz von DBT
FND(Find)	Suchen von Zeichenketten in Textdateien
FNT	Auswahl von Schriftarten des Epson-FX-80 und kompatible Drucker
FMT(Format)	Formatieren (Druck aufbereiten) von Textdateien
LST(List)	Ausgabe von Textdateien auf den Bildschirm
MRG(Merge)	Zusammenhängen von zwei sortierten Textdateien
PRT(Print)	Drucken von Textdateien
SRT(Sort)	Sortieren von Textdateien
TRN(Trans)	Kopiert Textdateien und ändert Zeichenketten

Tabelle 2. Dienstprogramme für Textverarbeitung und Dateiverwaltung

Assembler MAC ist ein 2-Pass-Compiler zur Erzeugung von verschiebbarem Objektcode. Der erzeugte Code liegt im Microsoft-8-Bit-Format zur Weiterverarbeitung mit dem Linker LNK oder L80 von Microsoft vor. Diese Linker verknüpfen einzeln übersetzte Module oder solche, die in einer Bibliothek stehen, zu einem lauffähigen Programm, einer sogenannten COM-Datei. Auch der Lader ist ein nützliches Hilfsmittel. Er ermöglicht es, Programme an eine bestimmte Adresse in den Speicher zu laden und wahlweise zu starten. Damit werden Betriebssystemerweiterungen beim Booten (Kaltstart) installiert. Der Bibliotheksverwalter (LIB) verwaltet eine Sammlung von verschiebbaren Objektmodulen. In der Tabelle 1 sehen Sie eine Übersicht aller vorhandenen Funktionen. Die Verwaltung geschieht mit Hilfe einer Indexdatei. In dieser Datei stehen die Namen und die Adressen der Module aus der Bibliothek. Wird nun ein Programm mit dem Linker zusammengebunden, so sucht der Computer zunächst den Namen in der Indexdatei. Bei einem positiven Suchergebnis wird das Objektmodul aus der Bibliothek an das lauffähige Programm angehängt. Der Assembler arbeitet mit Maschinen-Instruktions-Tabellen, um einen Quellcode zu übersetzen. Um den Assembler für verschiedene Prozessoren (8080/Z80) zu verwenden, muß ihm eine solche Tabelle zur Verfügung stehen. Das Programm CMIT paßt den Small-MAC-Assembler einem bestimmten Prozessor an, indem er eine Tabelle (es sind für beide Prozessortypen die Tabellen in Quellform vorhanden) in ein internes Format verwandelt und dieses in den ausführbaren Small-MAC-Assembler kopiert. Das Hilfsmittel DREL erzeugt aus jedem Objektmodul eine Liste in hexadezimalem Format. Die Ausgabe erfolgt im Standardformat. Es kann damit nach Belieben in eine Datei, auf einen Drucker oder auf den Bildschirm ausgegeben werden.

Neben dem C-Compiler und dem Assembler enthält das Entwicklungssystem noch viele andere nützliche Programme. Sie umfassen folgende Funktionen und können nach Belieben vom Anwender erweitert werden:
- editieren, formatieren, sortieren, zusammenfügen, listen, drucken, suchen, ersetzen, übersetzen, kopieren und aneinanderfügen, verschlüsseln und entschlüsseln, Leerzeichen durch Tabs ersetzen, Tabs durch Leerzeichen ersetzen, Zeichen, Wörter und Zeilen zählen, Druckerzeichensatz auswählen.

Tabelle 2 listet alle vorhandenen Programme auf. Die Dienstprogramme (insgesamt 15) werden nur als Quelldatei geliefert. Sie müssen also erst mit Small-C übersetzt und dann assembliert und zusammengefügt werden.

Außer den Quellen der Programme befinden sich noch verschiedene Include-Dateien auf den insgesamt drei Disketten, die man beim Kauf bekommt. Alles in allem beläuft sich die Anzahl der Programme auf den Disketten auf zirka 100 Dateien. Das deutsche Handbuch ist mit 200 Seiten sehr umfangreich und geht auf alle Programme sehr ausführlich ein. Die umfassenden Kommentare bewahren auch den Anfänger vor großen Problemen.

Das Entwicklungssystem Small-C bietet, trotz des eingeschränkten Sprachumfangs und seiner durch das Betriebssystem CP/M bedingten langsamen Diskettenverarbeitung, ein sehr gutes Preis/Leistungsverhältnis. Es gewährt vor allem dem C-Einsteiger hilfreiche Unterstützung beim Programmieren beziehungsweise beim Erlernen von C. Das Small-C wird zusammen mit C-Quellcode, Editor, Assembler, Linker und Tools zur Textverarbeitung für den C 128 und 128 D, den Schneider CPC 464/664/6128 und den Joyce zum Preis von 148 Mark von Markt & Technik angeboten. Besitzer des CPC 464/664 benötigen allerdings noch eine Speichererweiterung.

(Günter Langheinrich/cg)

Forth – die etwas andere Programmiersprache

Wenn man Forth lernen möchte, braucht man dazu zweierlei: ein Forth-System und ein Buch zum Lernen. Der folgende Artikel soll Ihnen den Einstieg in Forth lediglich erleichtern; wir sagen Ihnen, welche Literatur geeignet ist und worauf Sie bei einem Forth-Compiler für Ihr Computersystem achten müssen.

Sicher haben Sie schon den einen oder anderen Artikel über Forth gelesen. Dort war häufig die Rede von der »umgekehrt polnischen Notation«, vom Stackkonzept und von Worten wie SWAP, DUP und ROT, die dem Uneingeweihten allenfalls ein Stirnrunzeln entlocken. Über das Wesen der Sprache sagen sie aber nichts aus. Warum sich Forth in letzter Zeit dennoch zum »Geheimtip« unter Programmierern entwickelte, hat andere Gründe. Forth ist sicher nicht die »eierlegende Wollmilchsau«, wie dies von manch anderen Programmiersprachen behauptet wird, bietet aber einige Vorteile, die eine genauere Betrachtung rechtfertigen. Eine Warnung vorweg: Forth ist in vieler Hinsicht ungewöhnlich und sicher nicht jedermanns Sache. Wenn Sie Angst vor den Innereien Ihres Computers oder Systemabstürzen haben, dann ist Forth nichts für Sie. Arbeiten Sie jedoch gern maschinen-

nah, um die letzten Feinheiten aus Ihrem Computer herauszuholen, und können Sie mit Bits und Bytes etwas umgehen, dann lesen Sie weiter.

Allen höheren Programmiersprachen ist eins gemeinsam. Damit der Prozessor, das Herzstück eines jeden Computers, versteht, was der Programmierer will, muß der Programmtext übersetzt werden. Letztlich kann der Computer immer nur seinen Maschinencode verstehen.

Programmiersprachen lassen sich in zwei Kategorien einteilen. Da sind auf der einen Seite die Interpretersprachen wie Basic, Logo und Comal. Bei diesen Sprachen wird ein Programm während der Ausführung übersetzt (interpretiert). Dieses Verfahren hat den Vorteil der Interaktion, das heißt, man kann die Wirkung eines Befehls oder einer Befehlsfolge sofort sehen. Programme entwickeln geht also verhältnismäßig schnell, und auch die unvermeidliche Fehlersuche ist kein Problem. Man läßt einfach sein Programm Schritt für Schritt ablaufen, um zu sehen, wann und wo der Fehler auftritt. Nachteil dieser Methode ist jedoch die geringe Ablaufgeschwindigkeit der Programme. Schließlich beschäftigt sich der Computer den größten Teil seiner Rechenzeit mit der Übersetzung und nicht mit der Bearbeitung des eigentlichen Programms. Jeder, der einmal ein längeres Basic-Programm mit einem entsprechenden Programm in Maschinencode verglichen hat, wird dies bestätigen.

Aus diesem Grund schlägt man bei den Compilersprachen wie Pascal, C oder Modula einen anderen Weg ein. Der Programmtext wird ein einziges Mal in Maschinencode übersetzt (compiliert) und kann dann immer wieder ausgeführt werden. Das klingt gut, hat aber natürlich ebenfalls seine Tücken. So ist es ein langer Weg vom Quelltext bis zum lauffähigen Programm. Typisch für solche Sprachen ist der Dreischritt Editor, Compiler, Linker, der sich immer wiederholt und viel Zeit kostet. Ein Programmfehler wird ja fast immer erst ganz am Schluß erkannt, wenn der Computer sang- und klanglos abstürzt.

Forth verbindet die Vorteile von Interpreter- und Compilersprachen, also hohe Ablaufgeschwindigkeit und interaktive Programmentwicklung. In Forth können Sie die Wirkung jedes Befehls wie in Basic unmittelbar überprüfen, die Programme laufen jedoch zirka zehnmal schneller ab.

Zudem steht es offen, zeitkritische Programmteile mit dem fast immer vorhandenen Assembler unmittelbar in Maschinencode zu schreiben. Solche Routinen sind in der Regel sehr kurz. Der Forth-Interpreter/Compiler behandelt sie aber genauso wie alle anderen Forth-Befehle auch; Sie stoßen also auf keine SYS- oder CALL-Sequenzen mit irgendwelchen unverständlichen Zahlen dahinter.

Forth ist vollkommen strukturiert. Ein Programm besteht aus einer Reihe von Befehlen, in Forth Wörter genannt, die jeweils einzeln entwickelt und getestet werden. Jeder so neu erzeugte Befehl wird durch seine Definition dem Sprachkern hinzugefügt und steht damit zur Bildung weiterer Wörter bereit. Für Verknüpfungen stehen die von Pascal bekannten Kontrollstrukturen wie IF..ELSE..THEN, BEGIN..WHILE ..REPEAT, DO..LOOP und so weiter zur Verfügung. GOTO und GOSUB fehlen ebenso wie Zeilennummern. Die Wörter werden einfach durch Nennung ihres – hoffentlich sinnvollen – Namens aufgerufen. Das Ergebnis ist ein übersichtlicher, wartungsfreundlicher Code. Das lernt man spätestens dann zu schätzen, wenn man sich nach einigen Wochen oder Monaten ein Programm zur Überarbeitung wieder vornimmt. Haben Sie das einmal mit einem schlecht dokumentierten Assemblerprogramm versucht, wissen Sie das zu schätzen.

In der Kürze liegt die Würze

Forth macht nahezu alles möglich, insbesondere den Zugriff auf die gesamte Hardware. Viele Sprachen schieben da einen Riegel vor, entscheiden für den Programmierer, was erlaubt ist und was nicht. Forth ist hier genauso wie Assembler und bietet übrigens auch die gleichen Fehlerquellen. So sind Ein-Ausgabe-Bausteine unter Forth ebenso leicht zu programmieren, wie man sich eigene Speicherverwaltungen abseits von gewöhnlichen Variablen oder Arrays zusammenbauen kann. Ja, sogar der Forth-Compiler ist offen für Veränderungen. Dieser uneingeschränkte Zugang verlangt natürlich viel Disziplin beim Programmieren. Allerdings: Mehr als abstürzen kann Ihr Computer auch unter Forth nicht.

Forth-Programme sind kurz, in der Regel sogar kürzer als entsprechende Assembler-Programme. Das liegt an der Arbeitsweise des Forth-Compilers. Im Speicher steht bei jedem Wort nur eine Liste mit Adressen der Worte, aus denen es sich zusammensetzt. Der innerste Kern des Forth-Interpreters – übrigens eine Maschinencodesequenz von zirka 10 Byte – sorgt für die Abarbeitung dieser Liste. Das ist die Grundidee. Natürlich gibt es eine Reihe von Feinheiten. So kommt es, daß »normale« Forth-Systeme mit 300 bis 400 Befehlen Grundwortschatz nur zirka 10 bis 20 KByte Speicher benötigen.

Nachdem wir Ihnen nun den Mund hoffentlich ausreichend wäßrig gemacht haben, wenden wir uns wieder der eingangs gestellten Frage zu. Neben einem Computer braucht man, so hieß es dort, ein Forth-System und Literatur. Die Frage nach der Literatur läßt sich verhältnismäßig leicht beantworten. Es gibt ein für Anfänger wie Fortgeschrittene gleichermaßen geeignetes Buch. Es heißt »Starting Forth« von Leo Brodie. Eine deutsche Übersetzung ist unter dem Namen »Programmieren in Forth« im Hanser-Verlag erschienen. Das Buch entwickelte sich zu so etwas wie einem Standardwerk, weil es locker und doch exakt geschrieben ist. Wer gut Englisch kann, dem sei auf jeden Fall die Originalausgabe empfohlen, weil sich in der deutschen Übersetzung, vor allem bei den Programmbeispielen, einige Fehler eingeschlichen haben. Man sollte es allerdings nicht abends im Bett kurz vor dem Einschlafen lesen, sondern neben den Computer legen und damit arbeiten. Auch Forth lernt man, indem man in Forth programmiert und nicht beim Lesen.

Womit wir bei der Frage eines Forth-Systems wären. Hier läßt sich natürlich keine allgemeingültige Lösung angeben, da es für jeden Computer verschiedene Versionen gibt. Vor einiger Zeit veröffentlichte die Zeitschrift »Forth-Dimensions« (von der amerikanischen »Forth Interest Group« herausgegeben), eine Art Checkliste für Forth-Systeme. Wir bringen zu Ihrer Orientierung eine deutsche Übersetzung, damit Sie wissen, worauf man achten sollte. Maximal sind dreizehn Punkte zu vergeben. Systeme mit weniger als sieben Punkten dürften Ihnen die Arbeit mehr erschweren als erleichtern, von ihnen sollten Sie lieber die Finger lassen.

Die Größe eines Systems entscheidet über den Umfang, also die Anzahl der Befehle, die im Grundwortschatz enthalten sind. Sicher ist die reine Länge in KByte kein ausreichendes Merkmal, wichtiger ist die Befehlsanzahl. Sie gibt aber doch einigen Aufschluß, ob es sich um eine reine Standardimplementation handelt, die mit zirka 10 KByte auskommt, oder ob etwas Komfort geboten wird. Auch läßt sich aus der Länge in etwa beurteilen, wie viele der Grundworte in Maschinencode geschrieben sind. Zwar etwas länger, wirken sie sich aber positiv auf die Laufzeit des Systems aus. Trotzdem: Seien Sie vorsichtig, wenn Ihnen Systeme mit 30 KByte und mehr Länge angeboten werden. Diese enthalten normalerweise viel überflüssigen Ballast.

Viel wichtiger ist dagegen der nächste Punkt. Viele Systeme enthalten



Zusätze in Form von Quelltexten. Das hat zwei Vorteile. Zum einen muß man nur die wirklich benötigten Teile laden, zum anderen kann man die Quelltexte seinen Wünschen anpassen und verändern. Nebenbei bemerkt lernt man bekanntlich aus Beispielen am besten: Wem solche Beispiele gleich mitgeliefert werden, der hat es einfacher. Zusätze enthalten Programmierhilfen wie Decompiler, Einzelschritttracer, Assembler und so weiter oder auch fertige Programme.

Wie steigt man ein?

Die nächsten beiden Punkte beziehen sich auf den Editor. Wir halten dies für eins der wichtigsten Hilfsmittel jeder Programmiersprache. Schließlich arbeiten Sie beim Programmieren fast ausschließlich mit dem Editor. Sein Komfort entscheidet über die Bedienbarkeit eines Systems. Entspricht der Editor dem in Starting Forth beschriebenen, hat das den Vorteil, daß Sie die Beispiele aus dem Buch leichter bearbeiten können. Ein guter Editor sollte bildschirmorientiert arbeiten, das heißt, man kann mit dem Cursor auf dem Bildschirm »umherwandern« und Änderungen sofort sehen. Der Starting-Forth-Editor verfügt über diese Eigenschaften von Haus aus nicht, es gibt aber entsprechend erweiterte Versionen. Ein zeilenorientierter Editor ist nur zu empfehlen, wenn man noch nie mit etwas anderem gearbeitet hat.

Zu den nächsten beiden Punkten: Alljährlich treffen sich Forth-Programmierer, die diese Sprache professionell nutzen, in Amerika, um über Veränderungen oder Erweiterungen des Forth-Sprachkerns zu beraten. Dabei werden die Erfahrungen, die sich aus der praktischen Arbeit mit Forth ergaben, aufgearbeitet und gleichzeitig neue Ansätze diskutiert. Wenn Einigkeit über eine notwendige Änderung herrscht, wird ein neuer Standard festgelegt. Der letzte Standard wurde 1983 beschlossen. Davor gab es einen im Jahr 1979. Zwar erscheinen die Änderungen – vor allem dem Anfänger – häufig spitzfindig. Sie sorgen jedoch für eine stetige Weiterentwicklung der Sprache und verhindern das Auseinanderfallen in verschiedene Dialekte, wie es zum Beispiel bei Basic geschehen ist. Ein Programm, das nur Standard-Definitionen enthält, hat den unschätzbaren Vorteil, daß es unabhängig vom verwendeten Computer läuft. So lassen sich Programme für die verschiedensten Computer untereinander austauschen, jeder kann von Erfahrungen anderer profitieren. Die folgenden Punkte beziehen sich auf die Beschreibung und Unterstützung Ihres Systems. Eine Beschreibung ist unerläßlich, denn jedes Forth-System verfügt über eine Reihe von systemspezifischen Erweiterungen. Der Standard legt nur zirka 150 Worte fest, ein übliches Forth-System enthält aber 300 bis 400 Worte. Darüber hinaus sind häufig Besonderheiten und Zusätze eingebaut, die man natürlich nur mit einer vernünftigen Beschreibung ausnutzen kann. Diese sollte auch Informationen darüber enthalten, wie der Compiler arbeitet, welchen Speicher das System benutzt und so weiter. Je mehr Sie über Ihr Forth-System wissen, desto besser, auch wenn Sie mit den Informationen zunächst nicht viel anfangen können. Von einem guten Händler können Sie auch erwarten, daß er auf Ihre Fragen und Probleme eingeht.

Die letzten Punkte der FIG-Checkliste behandeln spezielle Erweiterungen. Ein Assembler sollte zur Grundausstattung jedes Systems gehören; ein Fließkommapaket wird man dagegen nur selten brauchen. Forth arbeitet aus Geschwindigkeitsgründen fast ausschließlich mit Integerarithmetik. Sogar trigonometrische Funktionen, wie man sie für grafische Anwendungen häufig braucht, lassen sich ohne Fließkomma programmieren. Zugriff auf ein File-System sollte möglich sein, weil es sonst schwierig wird, von Forth aus auf Files einer Textverarbeitung oder einer Tabellenkalkulation zuzugreifen. Forth selbst arbeitet normalerweise direkt auf Diskette, mit physikalischem Zugriff ohne File-System.

Wo bekommt man ein Forth-System?

Die verschiedensten Firmen bieten Forth an, zum Teil zu horrenden Preisen. Dabei muß der Preis kein Qualitätsmerkmal sein, im Gegenteil: Einige der besten Forth-Compiler sahen ihre Autoren als »public domain« (also der Allgemeinheit kostenlos zugänglich), was manche Vertreiber nicht davon abhält, sich die Anpassung auf ein spezielles Computersystem teuer bezahlen zu lassen. Den geringen Umsatz versucht man dann über hohe Preise auszugleichen.

Es gibt in Deutschland einen »Ableger« der »Forth Interest Group«, die »Forth Gesellschaft e.V.« in Hamburg. Sie setzt sich als Aufgabe die Verbreitung der Programmiersprache Forth. Die Gruppe arbeitet nicht kommerziell, sondern finanziert sich aus Beiträgen und Spenden. Dort kann man sich über Forth-Systeme für die verschiedenen Computer informieren, auch werden Bezugsquellen genannt. Es existiert auch eine Sammlung der verschiedensten Artikel über Forth, die man sich gegen Unkostenerstattung kopieren lassen kann. Für Mitglieder erscheint eine Zeitung namens »Vierte Dimension«, die zweimonatlich erscheint. Wer sich an die Forth-Gesellschaft wenden möchte, kann dies unter folgender Adresse tun:

Forth-Gesellschaft e.V.
Schanzenstr. 27
2000 Hamburg 6

Zum Abschluß noch ein Bonbon: Für den Commodore 64 und den Atari ST gibt es das »volksFORTH-83«, ein Forth-System, das von Mitgliedern der Forth-Gesellschaft geschrieben wurde und ebenfalls »public domain« ist. Es handelt sich um eins der besten Forth-Systeme, die es gegenwärtig gibt: Es entspricht vollständig dem 83'er-Standard, enthält Fullscreen-Editor, Assembler und eine Fülle von Tools, angefangen vom Decompiler bis hin zu einem Grafik-Paket. Das System ist multi-tasking-fähig, das heißt mehrere Programme können gleichzeitig ablaufen. Der Quelltext ist einschließlich des System-Quelltextes verfügbar und das System frei kopierbar. Die Weitergabe ist sogar ausdrücklich erwünscht. Wen es interessiert, kann es auch bei der Forth-Gesellschaft zu einem Selbstkostenpreis von 45 bis 65 Mark kaufen; man erhält dann – je nach Computersystem – mehrere Disketten mit sämtlichen Quelltexten sowie ein 200-seitiges Handbuch. Derzeit sind Versionen für Z80-, 8080- und 8068-Prozessoren in Arbeit. Nach der FIG-Checkliste erhält »volksForth-83« 12 von 13 möglichen Punkten.

Ebenfalls erwähnen wollen wir das F83 von Henry Laxen und Michael Perry. Auch hierbei handelt es sich um ein »public domain«-System nach dem 83'er-Standard. Es sind auch sämtliche Quelltexte zum System und allen Erweiterungen zu erhalten. F83 gibt es für 8080-, 8086- und 68000-Prozessoren. Bei der Forth-Gesellschaft erhalten Sie auf jeden Fall eine Version für den IBM-PC und kompatible Rechner. Zum F83 liegt keine Dokumentation vor. Statt dessen enthält die Version für jeden Quelltext Kommentarscreens. Auch dieses System erhält laut Checkliste 12 Punkte.

Es steht nun nichts mehr im Wege, eigene Erfahrungen zu sammeln. Beschaffen Sie sich Starting Forth und ein Forth-System, und setzen Sie sich eine Woche oder sagen wir bis zum dreißigsten Systemabsturz an Ihren Computer. Entweder werfen Sie dann, dem Wahnsinn nahe, alles in die Ecke, oder aber die Faszination dieser Sprache hat Sie gepackt und läßt Sie so schnell nicht mehr los.

(Dietrich Weineck/hg)



Der Einstieg in Pascal

Sie wollten schon immer wissen, was es mit der vielgerühmten strukturierten Programmierung in Pascal auf sich hat? Oder besitzen Sie gar einen Pascal-Compiler für Ihren Computer? Dann nehmen Sie sich ein wenig Zeit, und studieren Sie folgenden Artikel.

Zur Eingabe eines Pascal-Programmes, zur Übersetzung und zur Fehlerkorrektur müssen Sie spezielle Hilfsprogramme benutzen, deren Bedienung natürlich an dieser Stelle nicht beschrieben werden kann. In diesen Fällen hilft ein Blick in die Handbücher, die mit jedem Compiler geliefert werden.

Die grundlegenden Eigenschaften der Spache Pascal lassen sich am einfachsten an einem konkreten Beispielprogramm erläutern (Listing 1). Am besten lesen Sie zunächst einmal den gesamten Programmtext durch. Überlegen Sie sich für jede Zeile, welche Bedeutung die Anweisungen haben könnten. Auch ohne Kenntnisse in Pascal werden Sie erkennen, daß dieses Programm vom Benutzer die Eingabe einer Folge von Zahlen erwartet, über die eine Summe gebildet und gedruckt wird.

Auf den ersten Blick fällt bereits die stufenförmige Einrückung der Zeilen auf. Sie ist jedoch neben den Kommentaren zwischen »(*« und »*)« das einzige an einem Pascal-Programm, das keinerlei Bedeutung bei der Übersetzung des Programms besitzt. Genauer gesagt, betrachtet der Compiler ein Pascal-Programm als eine lange Folge von Symbolen (Sonderzeichen und Namen), die beliebig durch Leerzeichen oder Zeilenenden getrennt sein können. Im Prinzip könnte man also ein Pascal-Programm als einen riesigen Bandwurmsatz in eine einzige Zeile schreiben. Andererseits sind die Einrückungen, Leerzeilen und Kommentare die einzige Orientierungshilfe für den menschlichen Leser, um die teilweise komplex geschachtelten Schleifen und Abfragen in einem Programm auf einen Blick zu erkennen, so daß man einen »guten« Programmierstil bereits am systematischen Layout erkennt.

Es folgt also, daß man sich bei der Programmierung nicht auf eine »logische« Einrückung des Textes verlassen darf, sondern vielmehr durch die Verwendung von Sonderzeichen und Interpunktionszeichen (Semikolon, Punkt, Komma) die Interpretation des Programmes steuern muß.

```
PROGRAM SUMME (INPUT, OUTPUT);
(* DAS ERSTE PASCALPROGRAMM ZUR UEBUNG *)
  CONST ANZAHL   = 4;
        SUMME, X : REAL;
        I        : INTEGER;
BEGIN
  WRITELN('Geben Sie bitte ',ANZAHL,' Zahlen ein!');
  SUMME:= 0.0;
  FOR I:= 1 TO ANZAHL DO
    BEGIN
      READ(X); SUMME:= SUMME + X
    END;
  WRITELN;
  WRITELN('Die Summe betraegt ',SUMME);
  WRITELN('Der Durchschnitt ist ',SUMME / ANZAHL)
END.
```

Listing 1. Dieses Pascal-Programm berechnet Summe und Querschnitt

```
PROGRAM Name (INPUT, OUTPUT);

CONST  Konstantendeklarationen

VAR    Variablendeklarationen

PROCEDURE / FUNCTION Unterprogramm

deklarationen

BEGIN
  Anweisung; Anweisung; ...
END.
```

Bild 1. Der prinzipielle Aufbau eines Pascal-Programmes

AND	FILE	NOT	TO
ARRAY	FOR	OF	TYPE
BEGIN	FORWARD	OR	UNTIL
CASE	FUNCTION	PACKED	VAR
CONST	GOTO	PROCEDURE	WHILE
DIV	IF	PROGRAM	WITH
DO	IN	RECORD	
DOWNTO	LABEL	REPEAT	
ELSE	MOD	SET	
END	NIL	THEN	

Tabelle 1. Reservierte Wörter in Pascal

Neben den Sonderzeichen besteht ein Programm aus Wörtern. Dabei unterscheidet man zwischen reservierten Schlüsselwörtern (Wortsymbolen) und frei wählbaren Namen. Schlüsselwörter haben eine feste syntaktische Bedeutung und können nicht als Namen zum Beispiel für Variablen (wie SUMME) verwendet werden. Tabelle 1 zeigt die kurze Liste der reservierten Schlüsselwörter in Pascal. In den folgenden Artikeln werden Sie die Bedeutung der meisten dieser Wörter kennenlernen.

Namen in Pascal bestehen aus einem Buchstaben, dem eine beliebige Reihe von Buchstaben oder Zahlen folgt. Jeder Pascal-Compiler berücksichtigt mindestens die ersten acht Zeichen eines Namens, so daß Sie Namen nicht wie in Basic auf zwei Zeichen Länge verstümmeln müssen. Da in Pascal nicht nur Variable, sondern auch Unterprogramme, Konstanten und Typen mit Namen versehen werden, sollte man bei der Namensgebung möglichst systematisch vorgehen, um aus diesen Namen auf die Bedeutung schließen zu können.

Beispiele für sinnvolle und erlaubte Namen sind: ANFANGSBUCHSTABE, ENDEZEICHEN, GRENZE, FEHLER.

Nachdem Sie jetzt einen Überblick über die Einzelteile (Symbole) eines Pascal-Programms besitzen, wenden wir uns nun dem Zusammenbau dieser Elemente zu einem vollständigen Programm zu. Jedes Programm hat die in Bild 1 gezeigte Struktur:
- Programmkopf
- Deklarationen
- BEGIN
- Anweisungen
- END

Im Rahmen dieses Artikels besteht jeder Programmkopf aus dem Schlüsselwort PROGRAM, einem Namen und der Parameterliste (INPUT,OUTPUT). Die Namen INPUT und OUTPUT zeigen an, daß im Programm Eingaben von der Tastatur und Ausgaben an den Bildschirm vorkommen können.

Der nachfolgende Deklarationsteil definiert alle Namen, die im Programm verwendet werden. Grundsätzlich gilt in Pascal die Regel, daß die Definition eines Namens im Programm-Text vor der Anwendung des Namens in einer anderen Definition oder in einer Anweisung stehen muß. Dadurch können Pascal-Programme in einem einzigen Durchlauf (one pass) compiliert werden, da vor jeder Anwendung eines Namens der Compiler alle Informationen über einen Namen bereits gelesen hat.

In dem Beispielprogramm in Listing 1 wird zum Beispiel eine Konstante ANZAHL mit der Deklaration

```
CONST ANZAHL = 4
```

definiert. Immer wenn im nachfolgenden Programm-Text der Name ANZAHL

auftritt, wird die Konstante 4 compiliert. Außerdem würde der Compiler eine Zuweisung an diese Konstante, beispielsweise mit »ANZAHL := 39«, als Fehler erkennen. Dies ist einer der vielen Vorteile der expliziten Deklaration von Namen: Der Compiler schützt den Programmierer vor der falschen oder doppelten Verwendung eines Namens. Wer sich schon einmal durch ein größeres fremdes Programm gekämpft hat, wird auch wissen, daß die Abfrage

```
IF ANZAHL=MAXIMALANZAHL THEN...
```

verständlicher ist als

```
IF A=49 THEN...
```

Nicht zuletzt erlaubt die Verwendung von Konstanten eine einfache Änderung bestehender Programme. Soll das Programm in Listing 1 später einmal acht Zahlen summieren, so genügt die einmalige Änderung der Konstanten ANZAHL, wodurch sowohl die Obergrenze der FOR-Schleife als auch die Division bei der Durchschnittsbildung angepaßt wird.

A	B	A AND B	A OR B	NOT A
FALSE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE	FALSE
TRUE	TRUE	TRUE	TRUE	FALSE

Tabelle 2. Wahrheitstabelle der logischen Operatoren

```
PROGRAM GESCHWINDIGKEITSTEST (INPUT, OUTPUT);
  CONST LETZTERDURCHLAUF = 10000; (* WIEDERHOLE SCHLEIFE SO OFT *)
  VAR X1       : INTEGER;
      X2       : REAL;
      ZAEHLER: INTEGER;
BEGIN
  WRITELN('INTEGER-Schleife gestartet!');
  X1:= 0;
  FOR ZAEHLER:= 1 TO LETZTERDURCHLAUF DO X1:= X1+1;
  WRITELN('INTEGER-Schleife beendet!');
  WRITELN('REAL-Schleife gestartet!');
  X2:= 0;
  FOR ZAEHLER:= 1 TO LETZTERDURCHLAUF DO X2:= X2+1;
  WRITELN('REAL-Schleife beendet!');
END.
```

Listing 2. Ein Geschwindigkeitsvergleich zwischen REAL und INTEGER

Das Programm SUMME enthält noch eine weitere Form von Deklarationen:

```
VAR SUMME, X: REAL;
    I       : INTEGER;
```

Hinter dem Schlüsselwort VAR werden alle Variablen des Programms mit ihrem Typ (zum Beispiel INTEGER) aufgeführt. Der Typ einer Variablen gibt an, welche Werte eine Variable annehmen darf (zum Beispiel Zahlen oder Zeichen). Während in Basic der Typ einer Variablen aus dem Namen hervorgeht (A$, A%, A), wird in Pascal der Name eines Typs bei der Deklaration angegeben. Zunächst wollen wir uns auf die vordefinierten Standard-Typen beschränken:

REAL: Dieser Typ umfaßt die reellen Zahlen (zum Beispiel +1.0, 0.0, -1.2, 2.3E-4).

INTEGER: Werte vom Typ INTEGER sind ganze Zahlen aus einem beschränkten Intervall. Typischerweise sind dies die Zahlen zwischen -32768 bis 32767.

CHAR: Eine Variable vom Typ CHAR kann ein einzelnes Zeichen, wie »A«, »!« oder »4« speichern.

BOOLEAN: Dieser Typ ist Ihnen sicherlich nicht ganz so vertraut wie die übrigen Standard-Typen. Er umfaßt nämlich nur die logischen Werte »wahr« (TRUE) und »falsch« (FALSE). So liefern zum Beispiel Vergleiche ein Ergebnis vom Typ BOOLEAN. Mit booleschen Werten kann man über die logischen Operatoren UND, ODER und NICHT (AND, OR, NOT) »rechnen«, und die Ergebnisse in einer Variablen speichern:

```
27=13          (FALSE)
12>4           (TRUE)
'A'<'B'        (TRUE)
'A'='a'        (FALSE)
(A=B) OR (X=Y)
RICHTIG := ERGEBNIS = LOESUNG
FALSCH  := NOT (RICHTIG)
```

Die letzten beiden Zeilen verwenden zwei boolesche Variablen RICHTIG und FALSCH, die mit

```
VAR RICHTIG, FALSCH: BOOLEAN;
```

deklariert werden müssen. Diese Beispiele machen auch den Unterschied zwischen dem Vergleichsoperator »=« und dem Zuweisungsoperator »:=« deutlich. Mit »ERGEBNIS = LOESUNG« wird der Inhalt der Variablen ERGEBNIS mit dem Inhalt der Variablen LOESUNG verglichen. Stimmen beide überein, so liefert der Vergleich das Ergebnis TRUE. Um einer Variablen einen neuen Wert zuzuweisen, verwendet man den Zuweisungsoperator »:=«. Typische Beispiele sind:

```
I:= I+1
ERGEBNIS:= SIN(X)+COS(X)
LOESUNG:= A*B - C*D + 23.0
```

Zur Verdeutlichung der Wirkung der booleschen Operatoren enthält Tabelle 2 eine sogenannte Wahrheitstabelle, die zum Beispiel angibt, daß falsch und wahr gleich falsch ist (FALSE AND TRUE = FALSE).



Warum unterscheidet man aber zwischen den Typen REAL und INTEGER? Für einen Computer ist es wesentlich einfacher, mit ganzen Zahlen zu rechnen oder zu zählen, als Fließkomma-Arithmetik mit reellen Zahlen durchzuführen. Außerdem lassen sich Zahlen vom Typ INTEGER wesentlich kompakter als REAL-Zahlen speichern. Das bedeutet für die Praxis, daß man nur in Ausnahmefällen Variablen vom Typ REAL deklariert, und zwar nur dann, wenn sehr große Zahlen verarbeitet oder Nachkommastellen dargestellt werden müssen. Bei einer genaueren

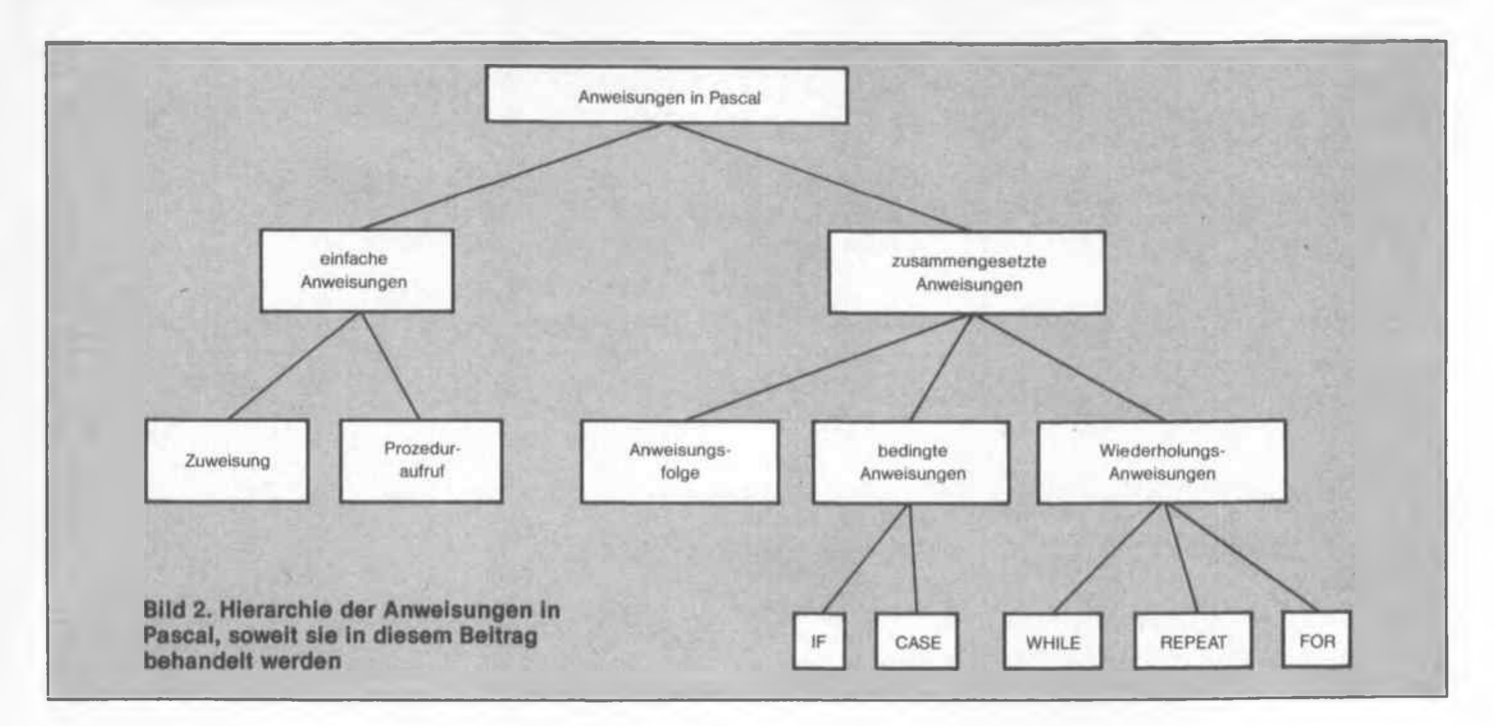


Bild 2. Hierarchie der Anweisungen in Pascal, soweit sie in diesem Beitrag behandelt werden

Untersuchung der meisten Programme wird man jedoch feststellen, daß die Mehrzahl der Variablen nur zum Zählen in Schleifen, als Indizes in Tabellen oder für ähnliche Steuerungsaufgaben benutzt werden, in denen INTEGER-Zahlen völlig ausreichen. Wer einen Computer zur Verfügung hat, sollte sich den erheblichen Geschwindigkeitsunterschied zwischen Operationen mit ganzen und reellen Zahlen am Beispielprogramm in Listing 2 verdeutlichen.

Zum Abschluß dieses kleinen Exkurses über die Standard-Typen in Pascal soll noch ein weiterer Begriff geklärt werden, der Ihnen bei der Lektüre eines Pascal-Lehrbuches oder in Fehlermeldungen des Compilers häufiger begegnen wird: Ein »skalarer Typ« ist ein Typ, dessen Werte einzeln aufzählbar sind und der nicht in einfachere Typen zerlegt werden kann.

INTEGER, CHAR und BOOLEAN sind beispielsweise skalare Typen. Reelle Zahlen oder Mengen haben keinen skalaren Typ. Später werden Anwendungen genannt, in denen nur Werte eines skalaren Typs auftreten dürfen. An solchen Stellen sind also nur ganze Zahlen, nicht aber reelle Zahlen zu verwenden.

Anweisungen und Blockstruktur

Jetzt können wir uns dem Anweisungsteil eines Pascal-Programms zuwenden, der definiert, was mit den Objekten geschieht, die im Deklarationsteil vereinbart wurden. Bild 2 zeigt einen Überblick über alle Arten von Anweisungen, die nachfolgend beschrieben sind.

Die Philosophie der sogenannten strukturierten Programmierung, die ja bekanntermaßen der Sprache Pascal zugrunde liegt, verbirgt sich in dem Zweig mit der Bezeichnung »Zusammengesetzte Anweisungen«. Jede zusammengesetzte Anweisung kann aus einer Vielzahl verschiedener (eventuell ebenfalls zusammengesetzter) Anweisungen bestehen. Man kann sich eine Anweisung als einen Block vorstellen, der einen Eingang und einen Ausgang besitzt. Ist ein Block eine zusammengesetzte Anweisung, so enthält er kleinere Blöcke mit je einem Eingang und Ausgang, die nach festen Regeln geschachtelt werden. So kann jeder Block als eine Einheit betrachtet werden, ohne die umgebenden Böcke zu kennen. Das mag etwas abstrakt klingen, aber das Bild einer Blockstruktur ist zur Verdeutlichung der Schachtelungen recht anschaulich.

Einfache Anweisungen: Dies sind die elementaren Bausteine, aus denen der Anweisungsteil besteht. Die wichtigste Form der einfachen Anweisungen hatten wir bereits kennengelernt: Eine Zuweisung besteht aus dem Namen einer Variablen, dem Zuweisungsoperator »:=« und einem Ausdruck. Das Programm in Listing 3 zeigt Ihnen eine Vielzahl von Beispielen für Zuweisungen in Pascal. Sie werden feststellen, daß sich die Ausdrücke von ihren Gegenstücken in Basic oder Fortran nur unerheblich unterscheiden. Sie haben also nicht eine so extravagante Form wie in C oder gar in Forth. Natürlich gilt auch in Pascal die Regel »Punkt- vor Strichrechnung«.

Bitte achten Sie darauf, daß der Typ des Ausdruckes auf der rechten Seite des »:=« mit dem Typ der Variablen vor dem »:=« verträglich ist. Daß man einer Variablen vom Typ CHAR keine Zahl zuweisen kann, ist wohl selbstverständlich. Jedoch ist es auch nicht möglich, einer Variablen vom Typ INTEGER direkt eine reelle Zahl zuzuweisen. Man

Fortsetzung auf Seite 30

```
PROGRAM ZUWEISUNGEN (INPUT, OUTPUT);
  VAR R: REAL;
      I: INTEGER;
      B: BOOLEAN;
      C: CHAR;
BEGIN
  R:= 3.141592653;                    (* PI ist eine reelle Zahl     *)
  R:= SQR(SIN(R)) + SQR(COS(R));      (* -> R = 1.0                  *)
  I:= 0;
  I:= I+1;                            (* vorwaerts zaehlen           *)
  I:= I*I + I - 7;                    (* Ergebnis ist ganze Zahl     *)
  I:= 13 DIV 4;                       (* Division mit Rest -> I=3    *)
  I:= 13 MOD 4;                       (* Divisionsrest -> I=1        *)
  R:= ABS(R);                         (* Absolutwert vom Typ REAL    *)
  I:= ABS(I);                         (* dsgl. vom Typ INTEGER       *)
  CH:='A';                            (* nur einzelne Zeichen !      *)
  CH:= CHR(65);                       (* CHR liefert das Zeichen     *)
                                      (* mit dem Code 65, das ist    *)
                                      (* der Buchstabe 'A'           *)
  I:= ORD('A');                       (* ORD wirkt genau umgekehrt,  *)
                                      (* also ist danach I= 65       *)
  B:= (I<>4);                         (* TRUE, falls I<>4            *)
  B:= B OR NOT B;                     (* Immer richtig (TRUE)!       *)
END.
```

Listing 3. Mögliche Formen der Zuweisung in Pascal

C-64



DER AUTOR RAETO WEST verwendete 1 Jahr der Analyse und Dokumentation auf den C-64! Ergebnis seiner völlig unzeitgemäßen Geduld: Das einzige enzyklopädische 64er-Buch, das neben Ihrem Computer liegen bleibt. Alle Erklärungen, auch komplexer System- und Programmfragen, umfassen bei Ray West stets beides: Kompetenz durch Einsicht und solides Faktenwissen. Beispielhaft: Musiktheorie und SID-Chip in Kapitel 13!

EIN REFERENZBUCH für professionelle Hard/Software-Entwickler auf dem US-Standard des Buchs PROGRAMMING THE PET/CBM des gleichen Autors; **EIN LEHRBUCH** zu Aufbau und Anwendung von Mikrocomputern am Beispiel des C-64 für alle Autodidakten und Einsteiger; **EIN ANWENDUNGS-HANDBUCH** zum C-64/SX-64 mit über 300 Programmierungen aller 64er-Funktionen – auch der schwierigen, seltenen und meist gemiedenen.

Beste Rezensionen in allen Zeitschriften.

te-wi Verlag GmbH
Theo-Prosel-Weg 1
8000 München 40

te·wi

688 Seiten, Softcover, DM 66,–

Weitere te-wi-Bücher



NEU! C-64 Akustik und Graphik
Ein planvoller Lehrgang – keine Beispielsammlung – in anschaulichem Stil – daher für jedes Alter. Dieses Werk eröffnet dem C-64 Benutzer die Welt der Graphiken und Klangbilder. Es enthält Programmbibliotheken und wird abgerundet durch zahlreiche Anhänge.
John Anderson, 208 Seiten, Softcover, DM 49,--



NEU!
Der Sensible C-64
C-64 Programmsammlung
Für Erstbenutzer wie für Experten – 2 Bücher der Softwarenutzung aller technologischen Eigenheiten des C 64.
Jedes Buch kostet DM 29,80



LOGO –
Jeder kann programmieren
(Daniel Watt)
Buch des Jahres in den USA. Für die Computer APPLE II, C-64, IBM PC, ATARI bis 520 ST., TI-99 und Schneider CPCs.
Hochwertiges Textbuch für Logo-Kurse für zu Hause und im Lehrbereich.
384 Seiten, A4, DM 59,–



NEU! Reparaturanleitung
Computer: C-64 Floppy: VC1541
Einzigartige Serviceunterlagen für Reparaturen und Entwicklungsarbeiten.
Enthält Schaltpläne, Bauteile- und Vergleichstypenliste, u. v. m; schnelle Servicetests; Anleitung zur systematischen Fehlersuche.
In A4-Mappe, je DM 29,80



STRUCTURED BASIC erweitert erheblich die Einsatzmöglichkeit des **C-64/C-128** auf Befehls- wie Speicherebene! Buch (376 S.) und Modul, DM 199,–
In Vorbereitung:
Die C-128 Enzyklopädie vom Erfolgsautor Raeto West. Ausgereift und in bewährter Solidität. Anfang 1986. Es lohnt sich zu warten. **ROM-Listing C-128** mit umfangreichen deutschen Kommentaren



Computer für Kinder
(Sally Greenwood Larson)
Ein Buch für Kinder und ihre Lehrer – ein kindgerechtes Buch für die erste Begegnung mit Computern, ihren Eigenwilligkeiten und ihren unerschöpflichen Möglichkeiten.
„Computer für Kinder" richtet sich an Kinder im Alter von 8 bis 13 Jahren. Ein Handbuch für Beginner. Unterhaltsam und leicht verständlich für die Computer **VC20** und **C-64.**
A4 quer. Je Ausgabe DM 29,80

tm 4416/6

Noch im Programm:

VisiCalc (mit CBM Diskette)	DM 79,--	C-64 IEEE-488 Buch und Steckmodul	DM 239,--
CBM Computer-Handbuch	DM 59,--	Umweltdynamik (Prospekt anfordern)	DM 59,--
Mikrocomputer-Grundwissen	DM 36,--	6502 – Programmieren in Assembler	DM 59,--

Fortsetzung von Seite 28

muß vielmehr angeben, was mit den eventuell vorhandenen Nachkommastellen geschieht. Hierzu dienen die Standardfunktionen TRUNC und ROUND, die Nachkommastellen einer reellen Zahl abschneiden oder zur nächsten ganzen Zahl aufrunden (Tabelle 3).

X =	TRUNC(X)	ROUND(X)
1.234	1	1
1.765	1	2
-1.234	-1	-1
-1.77	-1	-2

Tabelle 3. Die Wirkung von TRUNC und ROUND

Wie bestimmt man aber den Typ eines Ausdrucks? Dazu stellt man zunächst die Typen der Konstanten und Variablen (aus dem Deklarationsteil) fest. Eine Zahl ohne Dezimalpunkt und Exponent ist vom Typ INTEGER:
Beispiele:

```
1  0  -8  100000
```

(Zahlen vom Typ INTEGER)

```
1.2   1E-4    123.4
```

(Zahlen vom Typ REAL)

Wird eine reelle Zahl mit einer ganzen Zahl verknüpft, so erhält man als Ergebnis eine reelle Zahl. Indem man schrittweise (unter Beachtung der Prioritäten und Klammern) den Ausdruck auflöst, entsteht der Ergebnistyp.
Beispiele:

```
A:= 1000 * (1000 + 4)
B:= 1000 * (1000 + 4.0)
```

Da der Wertebereich INTEGER meist nur Zahlen kleiner oder gleich 32768 darstellen kann, würde bei der Ausführung der ersten Zuweisung ein Überlauf eintreten. Durch die Addition von 4.0 (vom Typ REAL) ist jedoch im zweiten Fall das Ergebnis (B) vom Typ REAL und somit noch darstellbar.

Tabelle 4 gibt einen Überlick über die möglichen Operatoren in Pascal mit konkreten Beispielen. Außergewöhnlich ist nur die Tatsache, daß die logischen Operatoren AND und OR stärker binden als die Vergleichsoperatoren (>,<,=,< >). Diese Einheit muß bei der Formulierung von Bedingungen (zum Beispiel bei IF-Anweisungen) durch eine korrekte Klammerung beachtet werden:
Beispiel:

```
IF (A=B) OR (C=D) THEN...
```

und nicht

```
IF A=B OR C=D THEN...
```

(Beim zweiten Ausdruck würde zunächst »B OR C« berechnet, das Ergebnis mit A verglichen, das Resultat (vom Typ BOOLEAN) schließlich mit D verglichen werden, was wohl nicht beabsichtigt war.)

Operator	Operation mit Beispiel	Operandentypen	Ergebnistyp
+ (Vorz.)	Identitaet + 3.4 (= 3.4)	INTEGER, REAL	wie Operand
- (Vorz.)	Vorzeichenwechsel - 3.4 (=-3.4)	INTEGER, REAL	wie Operand
+	Addition 1 + 1 (=2)	INTEGER, REAL	wie Operand
	Vereinigungsmenge	Mengen	Menge
	[1,2,3,4] + [3,5] (= [1,2,3,5])		
-	Subtraktion 1 - 1 (= 0)	INTEGER, REAL	wie Operand
	Differenzmenge	Mengen	Menge
	[1,2,3,4] - [3,5] (=[1,2,4])		
*	Multiplikation 1 * 1 (=1)	INTEGER, REAL	wie Operand
	Schnittmenge	Mengen	Menge
	[1,2,3,4] * [3,5] (= [3])		
DIV	Division mit Rest 7 DIV 3 (=2)	INTEGER	INTEGER
MOD	Divisionsrest 7 DIV 3 (=1)	INTEGER	INTEGER
/	'normale' Division 1 / 4 (=0.25)	INTEGER, REAL	REAL (immer)
=	gleich 'A'='a' [1,2,3]=[]	Skalar, Pointer	BOOLEAN
	'Otto' = 'Anna (alle FALSE)	Menge, String	
<>	ungleich (analog)	Skalar, Pointer	BOOLEAN
<	kleiner 'A'<'B' (= TRUE)	Skalar, String	BOOLEAN
>	groesser 'Otto'>'Anna' (=TRUE)	Skalar, String	BOOLEAN
>=	groesser oder gleich	Skalar, String	BOOLEAN
	Test auf Obermenge [1,2,3]>=[1] (= TRUE)	Menge	BOOLEAN
<=	kleiner oder gleich	Skalar, String	BOOLEAN
	Test auf Teilmenge	Menge	BOOLEAN
	[1,2,3,5] <= [1..4] (=FALSE)		
IN	Test auf Enthaltensein	Skalar und	BOOLEAN
	33 IN [1..40] (=TRUE)	Menge	
NOT	nicht NOT FALSE (=TRUE)	BOOLEAN	BOOLEAN
AND	und (I>J) AND (J>I) (=FALSE)	BOOLEAN	BOOLEAN
OR	oder (A=B) OR (A<>B) (=TRUE)	BOOLEAN	BOOLEAN

Tabelle 4. Operatoren und ihre Wirkungen

Erwähnenswert ist die Tatsache, daß es in Standard-Pascal keinen Exponentialoperator (A hoch B) gibt. Der weiterführende Artikel über Prozeduren und Funktionen beschreibt, wie man sich diesen Operator selbst definiert.

Um die Ergebnisse von Berechnungen und die Inhalte von Variablen am Bildschirm darzustellen und auch um Eingaben des Benutzers von der Tastatur zu lesen, gibt es in Pascal einige vordefinierte Unterprogramme. Diese Prozeduren werden durch die Angabe ihres Namens, den »Prozeduraufruf« aktiviert. Das Programm aus Listing 1 enthält bereits verschiedene Prozeduraufrufe. Mit WRITELN (write line) wird zum Beispiel der Cursor am Bildschirm auf den Anfang der nächsten Zeile gesetzt. Oft muß man einer Prozedur weitere Informationen übergeben. Diese »Parameter« werden in Klammern und durch Kommata getrennt hinter dem Prozedurnamen aufgeführt:

```
WRITE('SUMME = ', A+B+C);
WRITE(' DM')
```

Als Parameter für die Prozedur WRITE sind neben Textkonstanten (Strings in Hochkommata) alle Werte der Standard-Typen (REAL, INTEGER, CHAR und auch BOOLEAN) zugelassen. Gerade für Einsteiger bietet sich damit die Chance, während des Programmablaufs Zwischenergebnisse oder Variablen ohne großen Aufwand anzuzeigen. Zwei aufeinanderfolgende WRITE-Anweisungen drucken ihre Werte ohne Zwischenraum direkt hintereinander. Möchte man nach der Ausgabe einen Zeilenvorschub durchführen, so verwendet man WRITELN statt WRITE. Das Gegenstück zu WRITE und WRITELN sind die Prozeduren READ und READLN. Als Parameter an diese Prozeduren übergibt man Variablen, denen Werte zugewiesen sind, die von der Tastatur eingelesen werden. Betrachten wir folgendes Beispiel:

```
READ(A,B,C,D)
```

Dieser Prozeduraufruf liest vier Werte ein. Ob es sich dabei um ganzzahlige oder reelle Zahlen oder um Zeichen handelt, hängt von der Deklaration der Variablen A, B, C und D ab. Sind A, B, C und D Variablen vom Typ INTEGER, so kann der Benutzer folgende Eingaben machen:

```
1 2 3 4 [RETURN Taste]
```

oder auch in mehreren Zeilen:

```
1 2 [RETURN Taste]
3   [RETURN Taste]
4   [RETURN Taste]
```

Mit »READLN (A); READLN(B); READLN(C); READLN(D)« wird nach dem Einlesen jedes Wertes der Rest der Eingabezeile ignoriert, so daß die vier Werte in vier Zeilen eingegeben werden müssen.

Später werden Sie sehen, wie man eigene Prozeduren und Funktionen in Pascal definiert, denen man wie den Standard-Funktionen Parameter übergeben kann, oder die Ergebnisse zurückliefern. Solche benutzerdefinierten Prozeduren entsprechen also im Prinzip den Unterprogrammen (Stichwort GOSUB, RETURN) in Basic.

Jetzt sind alle einfachen Anweisungen bekannt, so daß wir mit der Besprechung der zusammengesetzten Anweisungen fortfahren können.

```
BEGIN (* TAUSCHE A <-> B *)
  H:= A;
  A:= B;
  B:= H
END
```

Listing 4. Ein Beispiel für einen abgeschlossenen Anweisungsblock

```
IF Bedingung THEN
  Anweisung
```

```
IF A=X THEN
  WRITELN(X, 'gefunden!')
```

Listing 5. Die einfache IF-Anweisung

```
IF Bedingung THEN
  Anweisung
ELSE
  Anweisung
```

```
IF A>B THEN
    MAXIMUM:= A
ELSE
    MAXIMUM:= B
```

Listing 6. Die IF-Anweisung mit ELSE-Zweig

Eine »Anweisungsfolge« faßt eine Reihe von Anweisungen zu einer einzigen Anweisung zusammen. Listing 4 zeigt als Beispiel für die allgemeine Struktur einer Anweisungsfolge ein kurzes Programmstück, in dem der Inhalt der Variablen A und B vertauscht wird. Dieses Diagramm soll zeigen, wie die kleinen Blöcke (Anweisungen) zu einem großen Block (der zusammengesetzten Anweisung) zusammengefaßt werden. Bitte beachten Sie, daß Semikola zwischen den Anweisungen stehen und nicht hinter jeder Anweisung. Daher muß vor dem abschließenden END kein Semikolon stehen. Jedoch ist es erlaubt, überflüssige Zeichen in eine Anweisungsfolge einzufügen, so daß folgende Anweisungsfolgen ebenfalls syntaktisch korrekt sind:

```
BEGIN H:=A; A:=B; B:=H; END
```

und

```
BEGIN ;H:=A;; A:=B; B:=H END
```

Der Anweisungsteil eines Pascal-Programms ist also syntaktisch gesehen eine Anweisungsfolge. Da die Anweisungen in einer Anweisungsfolge immer in derselben Reihenfolge abgearbeitet werden, benötigt man zur bedingten Ausführung von Befehlen eine zusätzliche zusammengesetzte Anweisung.

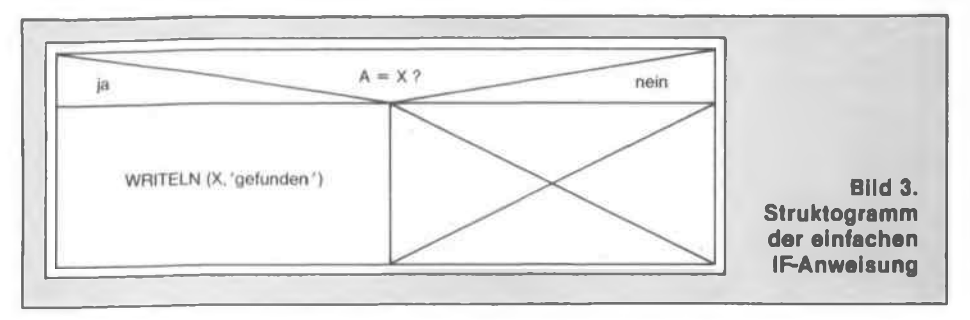


Bild 3. Struktogramm der einfachen IF-Anweisung

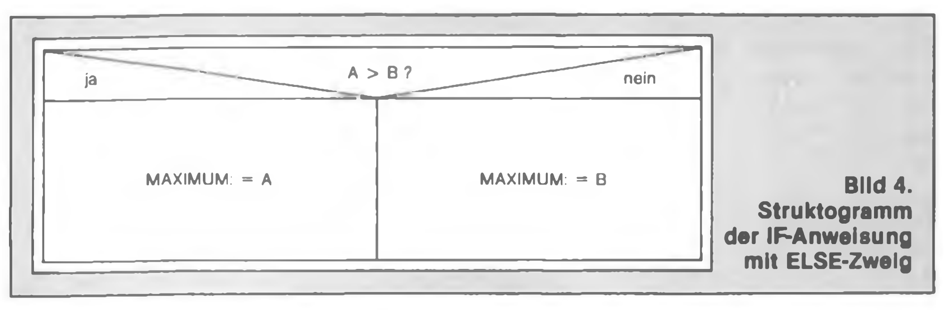


Bild 4. Struktogramm der IF-Anweisung mit ELSE-Zweig

```
PROGRAM PQFORMEL (INPUT, OUTPUT);
(* BERECHNE DIE LOESUNG DER GLEICHUNG X*X + P*X + Q = 0 *)
  VAR  P,Q,A,W: REAL;
BEGIN
  WRITE('P = '); READLN(P);
  WRITE('Q = '); READLN(Q);
  A:= - P / 2;
  W:= SQR(A) - Q;
  IF W<0 THEN
    WRITE('Es existiert keine Loesung')
  ELSE
    BEGIN
      WRITELN('X1 = ', A + SQRT(W));
      WRITELN('X2 = ', A - SQRT(W))
    END;
END.
```

Listing 7. »Quadratische Gleichung« als Beispiel für die IF-THEN-ELSE-Anweisung

Bedingungen auswerten mit IF

In Pascal gibt es zwei Formen der »IF-Anweisung«, die in Listing 5 und Listing 6 dargestellt sind. Die Bilder 3 und 4 sind die zu diesen Listings passenden Struktogramme. Die Bedingung nach dem Schlüsselwort IF ist ein beliebiger Ausdruck mit dem Ergebnistyp BOOLEAN. Ist der Wert des Ausdruckes TRUE, so wird die Anweisung hinter dem Schlüsselwort THEN ausgeführt. Ist das Ergebnis FALSE, so wird die Anweisung nach dem Schlüsselwort ELSE (falls vorhanden) ausgeführt.

Soll hinter THEN oder ELSE mehr als eine einzelne Anweisung stehen, müssen Sie diese Anweisungen mit BEGIN und END zu einer zusammengesetzten Anweisung »klammern«. Ein Beispiel hierfür zeigt Listing 7. Ein häufiger Fehler besteht darin, vor ELSE ein Semikolon einzufügen. In diesem Fall erkennt der Compiler die einseitige Auswahl

```
PROGRAM MAXIMUM (INPUT, OUTPUT);
  VAR MAX  : INTEGER;
      A,B;C: INTEGER;
BEGIN
  WRITE('A B C:'); READLN(A,B,C);
  IF A>B THEN
    IF A>C THEN
      MAX:= A
    ELSE
      MAX:= C
  ELSE
    IF B>C THEN
      MAX:= B
    ELSE
      MAX:= C;            (*<- erst hier darf ein Semikolon stehen! *)
  WRITELN('Das Maximum von ', A, B, C,' ist ',MAX)
END.
```

Listing 8. »Maximum« – ein Beispiel für verschachtelte IF-Abfragen

(wie in Bild 3) und »beschwert« sich, daß er mit dem Schlüsselwort ELSE nichts anzufangen weiß.

Natürlich kann die Anweisung nach THEN und ELSE wiederum eine IF-Anweisung sein, was in Listing 8 dazu verwendet wird, das Maximum der Zahlen A, B und C zu bestimmen. Bild 5 zeigt die zugehörige Blockstruktur, wobei man deutlich erkennt, daß jeder der geschachtelten Blöcke nur einen Eingang und einen Ausgang besitzt. Ein Beispiel für die Schachtelung der einseitigen Auswahl bietet Listing 9. Hier wird zu einem Tagesdatum das Datum des nachfolgenden Tages berechnet, wobei zur Vereinfachung angenommen wird, daß jeder Monat 30 Tage besitzt.

Wollen Sie komplexe IF..THEN..ELSE-Schachtelungen programmieren, so müssen Sie darauf achten, daß der Compiler jedes ELSE der letzten IF-

Anweisung zuordnet, die noch kein ELSE besitzt. Um in diesen Fällen Probleme zu vermeiden, empfiehlt es sich, die »innen« liegenden IF-Anweisungen mit BEGIN und END zu klammern.

Oft ist es erforderlich, in Abhängigkeit eines Wertes verschiedene Berechnungen durchzuführen. Für solche Fälle bietet sich die »CASE-Anweisung« an (Listing 10 und Bild 6). Es wird der Ausdruck nach dem Schlüsselwort CASE berechnet (der einen skalaren Typ besitzen muß) und in Abhängigkeit vom Ergebnis zu einer der Anweisungen hinter den Fallmarken verzweigt. Diese Fallmarken müssen jeweils Konstanten mit dem Typ des Ausdruckes hinter CASE sein. Stimmt keine der Fallmarken mit dem Ergebnis überein (zum Beispiel MONAT = 14), so erfolgt in Standard-Pascal ein Programmabbruch mit Fehlermeldung (viele Compiler erlauben jedoch auch die Angabe eines ELSE-Zweiges, auch OTHERWISE genannt, dessen Anweisung in diesem Fall ausgeführt wird).

Ein weiteres Beispiel für die CASE-Anweisung ist in Listing 11 gegeben. In diesem Programm werden von der Tastatur zwei Zahlen gelesen, die in Abhängigkeit von einem Zeichen (* . + - /) multipliziert, addiert, subtrahiert oder dividiert werden. Falls bei der Verarbeitung kein Fehler aufgetreten ist, wird das Ergebnis ausgedruckt. Bemerkenswert an dem Beispiel ist noch die Verwendung der booleschen Variablen OK, die den Wert FALSE enthält, falls ein illegaler Befehl eingegeben oder eine Division durch 0 versucht wurde.

In Basic würde man bei einem Fehler mit GOTO aus dem Inneren der CASE-Anweisung springen. Die auf den ersten Blick etwas schwerfällig anmutende Lösung mit der booleschen Variablen sorgt jedoch dafür, daß die CASE-Anweisung keinen zusätzlichen »Fehler-Ausgang« besitzt, der die Blockstruktur der zusammengesetzten Anweisung verletzen würde. Gerade bei großen Programmen sind nämlich Sonderbehandlungen mit Sprüngen quer durch den gesamten Programmtext eine schwer kontrollierbare Fehlerquelle.

Als kleiner Einschub ist in Listing 12 eine alternative Lösung der Aufgabe in dem Programm in Listing 10 gegeben. Sie nutzt die Tatsache, daß man in Pascal auch mit Mengen wie in der Mengenlehre in der Grundschule rechnen kann. Eckige Klammern ersetzen hierbei die geschweiften Klammern der Mathematik.

[1,2,3,4] und [1..4] bezeichnen dieselbe Menge, nämlich die Zahlen von 1 bis 4. Hingegen bezeichnen [] aber auch [3..1] die leere Menge, die keine Zahl enthält.

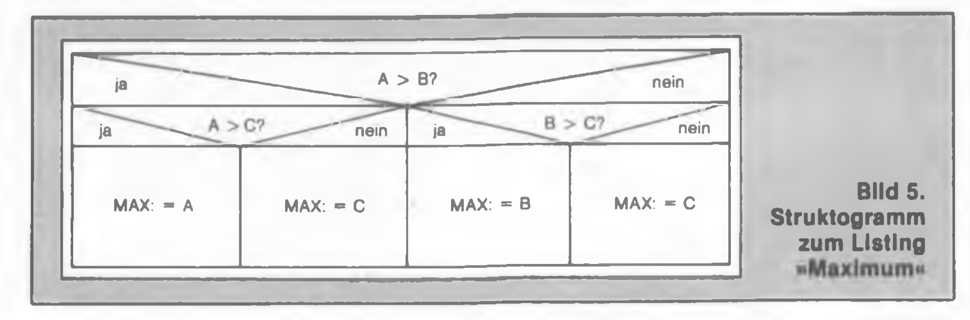


Bild 5. Struktogramm zum Listing »Maximum«

```
PROGRAM TAGESDATUM (INPUT, OUTPUT);
  CONST TAGE_PRO_MONAT  = 30;
        MONATE_PRO_JAHR = 12;
  VAR TAG, MONAT, JAHR: INTEGER;
BEGIN
  ...
  WRITELN('Heute ist der ', TAG, '.', MONAT, '.', JAHR);
  TAG:= TAG+1;
  IF TAG>TAGE_PRO_MONAT THEN
    BEGIN
      TAG:=1; MONAT:= MONAT+1;
      IF MONAT>MONATE_PRO_JAHR THEN
        BEGIN
          MONAT:=1; JAHR:= JAHR+1
        END
     END;
  WRITELN('Morgen ist der ', TAG, '.', MONAT, '.', JAHR);
END.
```

Listing 9. »Tagesdatum« – ein Beispiel zur Schachtelung der einseitigen Auswahl

```
CASE Ausdruck OF
  Fallmarke, ..., Fallmarke : Anweisung;
  Fallmarke, ..., Fallmarke : Anweisung;
  ...
  Fallmarke, ..., Fallmarke : Anweisung
END


CASE MONAT OF
  1,3,5,7,8,10,12: TAGE_PRO_MONAT:= 31;
  4,6,9,11       : TAGE_PRO_MONAT:= 30;
  2              : BEGIN
                     SCHALTJAHR:= (JAHR MOD 4) = 0) AND
                                 ((JAHR MOD 100)<>0) OR
                                  (JAHR MOD 400)=0);
                     IF SCHALTJAHR THEN TAGE_PRO_MONAT:= 29
                                   ELSE TAGE_PRO_MONAT:= 28
                   END;
END; (* CASE*)
```

Listing 10. Allgemeine Form und Beispiel zur CASE-Anweisung

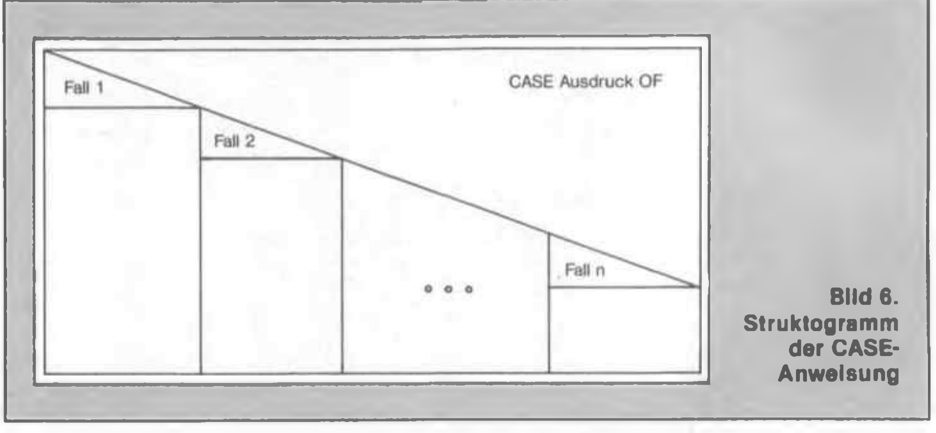


Bild 6. Struktogramm der CASE-Anweisung

Mit X IN [1,3,9,27]
wird geprüft, ob die Zahl X in der Menge der Dreierpotenzen bis 27 enthalten ist. In der Tabelle 4 auf Seite 30 sind einige dieser Operationen mit Mengen dargestellt (Vereinigungsmenge, Schnittmenge, Differenzmenge, Test auf Teilmengen, Test auf Gleichheit).

Bei der Besprechung der zusammengesetzten Anweisungen haben wir in der Übersicht (Bild 2) die »Wiederholungsanweisungen« erreicht. Es gibt in Pascal drei verschiedene zusammengesetzte Anweisungen, die Wiederholungen definieren, von denen sich jede für spezielle Anwendungsfälle besonders eignet.

Möchte man eine Anweisung so oft ausführen, bis eine gewisse Bedingung erfüllt wird, bietet sich die »REPEAT-Anweisung« (Listing 13) an. Zwischen den Schlüsselwörtern REPEAT und UNTIL steht eine beliebige Folge von Anweisungen, die Semikola trennen. Am Ende der Ausführung dieser Anweisungen wird der Ausdruck hinter dem

```
PROGRAM MINICOMPUTER (INPUT, OUTPUT);
  VAR CH       : CHAR;
      A, B, ERG: REAL;
      OK       : BOOLEAN;
BEGIN
  WRITELN('Geben Sie ein Operationszeichen (+,-,*,.,/) und zwei Zahlen ein!');
  READLN(CH, A, B);
  OK:= TRUE;              (* noch ist kein Fehler aufgetreten *)
  CASE CH OF
   '.','*'  : ERG:= A * B;
   '/'      : IF B = 0 THEN
                BEGIN OK:= FALSE;
                  WRITELN('Fehler: Division durch Null!')
                END
              ELSE
                ERG:= A / B;
    '+'     : ERG:= A + B;
    '-'     : ERG:= A - B;
   ELSE       BEGIN
                OK:= FALSE;
                WRITELN('Die Operation ', CH, ' ist nicht moeglich!')
              END;
   END; (* von CASE *)
   IF OK THEN
     WRITELN('Ergebnis: ', ERG)
 END.
```

Listing 11. Ein »Minicomputer« als weiteres Beispiel für CASE

```
IF MONAT IN [1,3,5,7,8,10,12] THEN TAGE_PRO_MONAT:= 31 ELSE
IF MONAT IN [4,6,9,11]        THEN TAGE_PRO_MONAT:= 30 ELSE
ELSE
  BEGIN (* hier ist MONAT = 2 *)
    SCHALTJAHR:= (JAHR MOD 4) = 0) AND
                ((JAHR MOD 100)<>0) OR
                 (JAHR MOD 400)=0);
    IF SCHALTJAHR THEN TAGE_PRO_MONAT:= 29
    ELSE TAGE_PRO_MONAT:= 28
  END;
```

Listing 12. Das Beispiel aus Listing 9, programmiert unter Verwendung von Mengen

```
REPEAT                              REPEAT
  Anweisung;                          WRITELN('Alles klar? (Ja oder Nein)');
  Anweisung;                          READ(CH)
  ....                              UNTIL CH IN ['J','j','N','n']
  Anweisung
UNTIL Bedingung
```

Listing 13. Allgemeine Form und Beispiel für REPEAT...UNTIL

```
PROGRAMM WURZEL (INPUT, OUTPUT);
  CONST EPS = 1.0E-7;     (* Genauigkeit: mindestens 7 Nachkommastellen*)
  VAR   X,Y,Z: REAL;
BEGIN
  WRITE('Die Quadratwurzel aus '); READ (X);
  IF X<0 THEN
    WRITELN(' ist keine reelle Zahl')
  ELSE
    BEGIN   (* 1. einen Startwert Z berechnen *)
      Y:= 2;
      REPEAT
        Z:= Y; Y:= Y*Y
      UNTIL Y>X;

      (* 2. Jetzt folgt die eigentliche Berechnung: *)
      REPEAT
        Y:= Z;
        Z:= 0.5 * (Y + X/Y)
       UNTIL ABS(Y-Z)<= EPS;       (* bis Abweichung kleiner als Genauigkeit *)

      WRITELN(' ist ', Z)
    END;
END.
```

Listing 14. Berechnung von Quadratwurzeln unter Verwendung von REPEAT...UNTIL-Schleifen

Schlüsselwort UNTIL ausgewertet. Er liefert einen booleschen Wert (TRUE oder FALSE). Ist die Abbruchbedingung nicht erfüllt, so wird die Anweisungsfolge wiederholt. In dem Beispiel werden also so lange Zeichen von der Tastatur eingelesen, bis der Benutzer ein »J« oder ein »N« eingegeben hat. Bei näherem Hinsehen können Sie also feststellen, daß Mengenoperationen auch auf Mengen von Zeichen anwendbar sind:

`CH IN ['j', 'J', 'n', 'N']`

Eine REPEAT-Schleife benutzt man bei Wiederholungen, die in der Abbruchbedingung einen Wert benötigen, der erst im Inneren der Schleife ermittelt wird. Als Beispiel ist in Listing 14 die Berechnung der Quadratwurzel für eine reelle Zahl X durch eine Iteration aufgeführt. Dieses Programm berechnet die Wurzel in zwei Schritten, die jeweils eine REPEAT-Schleife benötigen. Im ersten Schritt wird die kleinste Zweierpotenz (2,4,8,16,...) Z berechnet, die quadriert gerade größer als X ist. Anschließend wird in der zweiten REPEAT-Schleife dieser Startwert Z schrittweise modifiziert, bis die Differenz zwischen zwei Iterationswerten kleiner als die gewünschte Genauigkeit EPS ist.

Viel häufiger als die REPEAT-Anweisung findet die WHILE-Anweisung Verwendung (Listing 15 und Bild 8). Hier wird die Bedingung nach dem Wortsymbol WHILE vor der Ausführung der Anweisung nach DO geprüft. Damit besteht insbesondere die Möglichkeit, daß diese Anweisung kein einziges Mal ausgeführt wird, falls nämlich die Bedingung bereits beim Eintritt in die Schleife den Wert FALSE liefert.

Soll die WHILE-Schleife, was wohl in der Mehrzahl der Fälle zutreffen wird, mehrere Anweisungen umfassen, so muß man diese zu einer Anweisungsfolge mit BEGIN und END klammern. Ein kleines, aber besonders schönes Beispiel für die Funktionsweise der WHILE-Schleife zeigt Listing 16. Hier wird der Wert $E = N^K$ für natürliche Zahlen N und K berechnet. K gibt also an, wie oft N mit sich selbst multipliziert werden muß, um E zu erhalten. Daher wird im Programm die Zählvariable I verwendet, die von K abwärts gegen 0 zählt. Das Beispiel ist deshalb so ideal, da durch die Eigenschaft der WHILE-Anweisung, eine Prüfung am Beginn der Schleife durchzuführen, auch die Sonderfälle korrekt behandelt werden. In der Mathematik gilt nämlich:

$N^0 = 1$ für alle N

$0^K = 0$ für alle $K \neq 0$

Durch eine geeignete Wahl der Wiederholungsanweisung kann man sich also viele Sonderbehandlungen mit IF-Anweisungen ersparen.

Zum Vergleich ist in Listing 17 je eine WHILE- und eine REPEAT-Schleife angegeben, die alle Zahlen zwischen A und B druckt. Wenn Sie mit diesem Programm experimentieren, werden Sie feststellen, daß der einzige Unterschied bei der Ausführung darin besteht, daß in der REPEAT-Schleife für $A > B$ die Zahl A gedruckt wird, während die WHILE-Schleife in diesem Fall nicht in Funktion tritt.

Für eine Anwendung wie in Listing 17 wird man jedoch normalerweise die dritte Variante der Wiederholungsanweisungen in Pascal benutzen: Die FOR-Schleife (Listing 18) findet überall dort Anwendung, wo die Anzahl der Schleifendurchläufe vor dem Beginn der Schleife bereits bekannt ist. Den genauen Ablauf der Ausführung soll fol-

gendes Beispiel zeigen:
FOR I:= A TO B DO WRITE(I)

1. Berechne A und weise den Wert der Variablen I zu.
2. Berechne B und speichere den Wert in einer temporären (unsichtbaren) Variablen X
3. Ist I>X, so beende die Schleife
4. Sonst drucke die Zahl I
5. Erhöhe I um 1 und weiter bei 3

Die Variable I muß einen skalaren Typ besitzen (also zum Beispiel CHAR, aber nicht REAL) und mit dem Ergebnistyp von A und B verträglich sein. Aus dem obigen Schema (1..6) folgt außerdem, daß für A>B wie bei der WHILE-Schleife keine Zahl gedruckt wird. Man könnte in der Schleife die Variable B auch beliebig verändern, ohne die obere Grenze der Zählvariablen (die ja in einer temporären Variablen gespeichert wurde) zu beeinflussen.

Um rückwärts zu zählen, besteht noch die Möglichkeit, eine Variante der FOR-Schleife zu verwenden:
FOR I:= A DOWNTO B DO WRITE(I)

Die obigen Regeln gelten für diese Schleife analog. Mit der FOR-Schleife kann man nur in Einerschritten aufwärts oder abwärts zählen, andere Schrittweiten müssen mit einer WHILE-Schleife explizit programmiert werden. Im Beispiel aus Listing 19 wird ein Winkel PHI in Schritten von DELTA hochgezählt und dabei eine Funktion am Bildschirm »gezeichnet«.

Wenn Sie zu der Übersicht in Bild 2 zurückblättern, werden Sie erkennen, daß wir mit der Vorstellung der FOR-Schleife praktisch alle Anweisungen in Pascal besprochen haben. Zwei fundamentale Eigenschaften von Pascal wurden jedoch noch ausgespart.

(1) Zusammengesetzte Typen ermöglichen es, nicht nur mit einzelnen Zahlen und Zeichen, sondern sogar mit hierarchisch aufgebauten und auch veränderlichen Datenstrukturen zu arbeiten.

(2) Neben den bereits vorgestellten Standard-Prozeduren READ, READLN, WRITE und WRITELN gibt es noch eine Vielzahl an vordefinierten Prozeduren und Funktionen, die in jeder Implementation der Sprache Pascal vorhanden sind. Viel bemerkenswerter ist jedoch die Möglichkeit, beliebige Teile eines Programmes als Prozeduren und Funktionen zu definieren, die über bestimmte Schnittstellen mit ihrer »Umwelt« (Prozeduren oder Hauptprogramm) kommunizieren können.

In den folgenden Beiträgen werden diese beiden wichtigen und interessanten Gebiete ausführlich behandelt, wobei gleichzeitig das bisherige Wissen über die einfachen Typen und die Anweisungen in Pascal angewandt und vertieft wird. (Florian Matthes/ev)

```
WRITELN ('ALLES KLAR?')
READ (CH)
UNTIL CH IN ['J','j','N','n']
```

Bild 7. Struktogramm der REPEAT-Anweisung

```
WHILE Bedingung DO              WHILE X>=1.0 DO
  Anweisung                       X:= X / 2.0;
```

Listing 15. Allgemeine Form und Beispiel für die WHILE...DO-Schleife

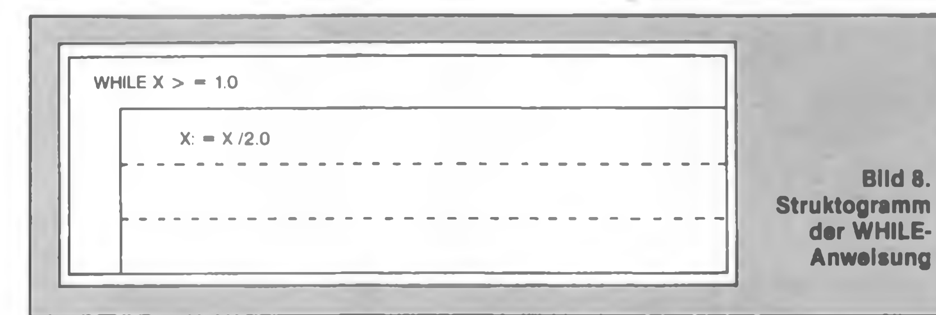


Bild 8. Struktogramm der WHILE-Anweisung

```
PROGRAM HOCH (INPUT, OUTPUT);
  VAR I, N, E, K: INTEGER;
BEGIN
  READLN(N, K);
  IF (N<0) OR (K<0) THEN
    WRITELN('ungueltige Eingabe!')
  ELSE
    BEGIN
      E:=1; I:=K;
      WHILE I>0 DO
        BEGIN
          E:= E*N;
          I:= I-1
        END;
      WRITELN(N, '^', K, '=', E)
    END
END.
```

Listing 16. Ein Demo-Programm unter Verwendung der WHILE...DO-Schleife

```
PROGRAM ZUM_VERGLEICH (INPUT, OUTPUT);
  VAR I, A, B: INTEGER;
BEGIN
  READLN(A, B);
  WRITE('Mit REPEAT:');
  I:=A;
  REPEAT
    WRITE(I);
    I:= I+1
  UNTIL I>B;
 WRITELN;

 WRITE('Mit WHILE:');
 I:= A;
 WHILE I<=B DO
   BEGIN
     WRITE(I);
     I:= I+1
   END;
 WRITELN
END
```

Listing 17. Ein Programm zum Vergleich zwischen WHILE und REPEAT

```
FOR Variable:= Ausdruck TO Ausdruck DO       FOR I:= 1 TO 1000 DO
  Anweisung                                    SUMME:= SUMME + 1 / I

FOR Variable:= Ausdruck DOWNTO Ausdruck DO   FOR I:= 1000 DOWNTO 1 DO
  Anweisung                                    SUMME:= SUMME + 1 / I
```

Listing 18. Allgemeine Form und Beispiel für die FOR-Schleife

```
PROGRAM SCHWINGUNG (INPUT, OUTPUT);
(* ZEICHE DIE FUNKTION F(PHI) = EXP(-PHI/PI) * SIN (PHI) *)
  CONST PI = 3.1415926;
        MAXZEILEN = 40;       (* Laenge der Ausgabe in Zeilen       *)
        MITTE = 40;           (* Mitte eines 80-Zeichen Bildschirms*)
  VAR   PHI, Y, DELTA : REAL;
        SPALTE, ZEILE: INTEGER;

BEGIN
  DELTA:= 6*PI / MAXZEILEN
  FOR ZEILE:= 0 TO MAXZEILEN DO
    BEGIN
      PHI:= DELTA * ZEILE;     (* AKTUELLER WINKEL *)
      Y  := EXP(-PHI/PI)*SIN(PHI);
      SPALTE:= MITTE + ROUND(MITTE * Y);

      WRITELN('*': SPALTE);    (* drucke Stern in Spalte SPALTE *)
    END;
END.
```

Listing 19. Schwingungs-Berechnung mit FOR-Schleife

Programmieren mit Pascal

Mit diesem Artikel steigen wir voll in die eigentliche Programmierung mit Pascal ein. Prozeduren, Funktionen und Datentypen heißen die Stichworte.

Wie steigt man am einfachsten in die Programmierung ein? Nun, vermutlich mit einem Programm zum Ausprobieren und Experimentieren. Das Beispielprogramm in Listing 1 liest einen Text von der Tastatur ein und zeichnet ein einfaches Balkendiagramm wie in Bild 1, das die Häufigkeit jedes Zeichens angibt. Wir möchten Sie bitten, vor dem Weiterlesen einen längeren Blick auf das Listing zu werfen, um einerseits die bereits im einführenden Artikel besprochenen Sprachelemente (zum Beispiel die FOR-Schleife) zu verstehen und andererseits einen Überblick über die neuen Probleme zu bekommen.

Der Kern des Programms besteht darin, für jeden Buchstaben einen Zähler zu führen, der bei jedem Auftreten des Buchstabens im eingelesenen Text erhöht wird. Natürlich möchte man nicht für jeden Buchstaben eine Zählvariable deklarieren und dann in einer riesigen CASE-Anweisung jeden einzelnen Zähler erhöhen. Statt dessen vereinbart man ein ARRAY (Feld, Tabelle) von Zählern:

```
VAR Z: ARRAY ['O'..'Z']
OF INTEGER;
```

Ein Array ist eine Variable, über deren Namen mehrere Werte des gleichen Typs angesprochen werden. Das Array Z besitzt für jedes Zeichen zwischen »0« und »Z« einen Wert vom Typ INTEGER. Auf die einzelnen Werte greift man durch Nennung des Array-Namens, gefolgt von einem Index in eckigen Klammern, zu:

```
Z['0']:= 13;
WRITELN (Z['X']);
Z[CH]:= Z[CH] + 1
```

Die letzte Zuweisung zeigt eine wichtige Eigenschaft von Arrays: Man kann als Index eine Variable (hier CH) oder einen beliebig komplexen Ausdruck verwenden. Für Basic-Programmierer ungewohnt ist die Eigenschaft von Pascal, nicht nur Zahlen, sondern beliebige geforderte Datentypen als Index zuzulassen.

Um alle Zähler in Z zurückzusetzen, kann man also folgende FOR-Schleife verwenden:

```
FOR CH:= '0' TO 'Z' DO Z[CH]:= 0;
```

Nach diesem speziellen Beispiel sollen Sie auch die allgemeinen Regeln für Arrays in Pascal kennenlernen:

1. Ein Array muß wie jede andere Variable im Variablendeklarationsteil vereinbart werden.

2. Ein Array ist ein zusammengesetzter Typ, der durch zwei Typen beschrieben wird: Den skalaren Indextyp und den beliebig wählbaren Elementtyp. Z hat also den Indextyp »'0' .. 'Z'« und den Elementtyp INTEGER.

3. Beim Indizieren muß der Typ des

```
PROGRAM HAEUFIGKEIT (INPUT, OUTPUT);
(* STATISTIK DER BUCHSTABENHAEUFIGKEIT IN EINEM TEXT *)

  CONST VON = '0'; BIS = 'Z';
        ENDEZEICHEN = '@';
        BILDSCHIRMBREITE = 80;          (* 80 ZEICHEN PRO ZEILE AM BILDSCHIRM *)

  TYPE  ZAEHLERFELD = ARRAY[VON..BIS] OF INTEGER;

  VAR   Z : ZAEHLERFELD;

  (* ---------------------------------------------- BEGINN DER PROZEDUREN *)

  PROCEDURE LOESCHEN(VAR A: ZAEHLERFELD);
  (* ALLE ZAEHLER ZURUECKSETZEN *)

    VAR CH: CHAR;

  BEGIN
    FOR CH:= VON TO BIS DO A[CH]:=0
  END; (* LOESCHEN *)

  PROCEDURE ZAEHLEN (VAR A: ZAEHLERFELD);
  (* TEXT VON DER TASTATUR BIS ENDEZEICHEN EINLESEN UND ZAEHLER IN A ERHOEHEN *)

    VAR CH: CHAR;

  BEGIN
    WRITELN('GEBEN SIE JETZT BITTE EINEN TEXT EIN, DER MIT ', ENDEZEICHEN,
           ' ENDET');
    REPEAT
      READ(CH);
      IF CH IN [VON..BIS] THEN
        A[CH]:= A[CH] + 1;
    UNTIL CH = ENDEZEICHEN;
    WRITELN
  END; (* ZAEHLEN *)

  PROCEDURE STATISTIK (A: ZAEHLERFELD);
  (* DRUCKE FUER JEDEN BUCHSTABEN EINEN BALKEN, DER DIE HAEUFIGKEIT DES      *)
  (* ZEICHENS ANGIBT.                                                        *)

    VAR CH   : CHAR;
        MAX  : INTEGER;
        HOEHE: INTEGER;

      PROCEDURE BALKEN (N:INTEGER);
      (* ZEICHNE EINEN HORIZONTALEN BALKEN MIT N ZEICHEN *)

       BEGIN
         WHILE N>0 DO
           BEGIN
             WRITE('#'); N:= N-1
           END;
         WRITELN;
       END; (* BALKEN *)

   BEGIN (* HIER BEGINNT DER ANWEISUNGSTEIL DER PROZEDUR STATISTIK *)
     WRITELN('STATISTIK DER BUCHSTABENHAEUFIGKEITEN:');

     (* BESTIMME ZUNAECHST DEN MAXIMALEN ZAEHLERSTAND: *)
     MAX:= A[VON];
     FOR CH:= SUCC(VON) TO BIS DO
       IF A[CH]>MAX THEN MAX:= A[CH];
    (* ZEICHEN JETZT DIE BALKEN: *)
    FOR CH:=VON TO BIS DO
      BEGIN
        WRITE(CH, ' |');
        HOEHE:= ROUND( A[CH] * (BILDSCHIRMBREITE-3)/MAX);
        (* DAMIT NIMMT DER HOECHSTE BALKEN DIE GESAMTE BILDSCHIRMBREITE EIN  *)
        BALKEN (HOEHE);
      END;
   END; (* STATISTIK *)

   (* ---------------------------------------------- ENDE DER PROZEDUREN *)

BEGIN
  LOESCHEN (Z);
  ZAEHLEN  (Z);
  STATISTIK(Z);
END.
```

Listing 1. Programm »Häufigkeit«

Ausdruckes in eckigen Klammern mit dem bei der Deklaration vereinbarten Indextyp verträglich sein. »Z[5]:=0« wäre nach der oben angegebenen Deklaration von Z nicht zulässig, da die Zahl 5 kein Zeichen ist. Außerdem ist auch die Zuweisung »Z['!']:=3« nicht erlaubt, da das Zeichen '!' nicht im Intervall »0« bis »Z« liegt.

Wenn Sie die Deklaration der Variablen Z im Beispiel 24 untersuchen, werden Sie feststellen, daß diese in drei Schritten erfolgte: Zunächst wurden zwei Konstanten

```
CONST VON='0'; BIS='Z';
```

eingeführt. Mit diesen beiden Konstanten erfolgte im nächsten Schritt eine Typendeklaration:

```
TYPE ZAEHLERFELD = ARRAY
[VON .. BIS] OF INTEGER;
```

Nach dem Schlüsselwort TYPE wird ein Name genannt, dem nach einem Gleichheitszeichen eine Typangabe folgt. Im dritten Schritt kann in der Variablendeklaration der Typbezeichner ZAEHLERFELD benutzt werden, um ein Array mit den Indexgrenzen VON und BIS und dem Elementtyp INTEGER zu definieren.

```
VAR A,B,FELD: ZAEHLERFELD;
     ZAEHLER: ZAEHLERFELD;
```

Neben der Einsparung von Schreibarbeit erhöhen Typnamen die Lesbarkeit von Programmen. Listing 2 zeigt einige Typdeklarationen und anschließend die Anwendung dieser Typnamen in weiteren Typ- und Variablendeklarationen.

Jetzt werden Sie wahrscheinlich alle Anweisungen mit den Array-Variablen A und Z verstehen. Neben der Verwendung von Arrays und Typnamen enthält das Programm »Häufigkeit« (Listing 1) noch ein weiteres neues Sprachelement von Pascal. Sie finden in dem Programm viermal das Schlüsselwort PROCEDURE, das eine Prozedurdeklaration einleitet. Nach diesem Schlüsselwort folgt der Name der Prozedur, über den die Prozedur aufgerufen wird. Der eigentliche Anweisungsteil des Programmes (nach dem letzten BEGIN) besteht also nur aus den folgenden drei Prozeduraufrufen:

```
LOESCHEN (Z);
ZAEHLEN  (Z);
STATISTIK(Z);
```

Das Programm besteht also aus drei voneinander unabhängigen Programmteilen zum Rücksetzen der Zähler, zum Einlesen des Textes und der Ausgabe der Statistik. Jeder Prozedur wird in Klammern das Array Z als aktueller Parameter übergeben.

Wie sieht aber eine Prozedurdeklaration aus? Prozedurdeklarationen müssen am Ende des Vereinbarungsteils (also hinter den Konstanten, Typ- und Variablendeklarationen) stehen. Die Prozedurdeklaration selbst besteht aus drei Teilen:

1. Der Prozedurkopf enthält nach dem Namen der Prozdur bei Bedarf in Klammern eine Liste der formalen Parameter:

```
PROCEDURE LOESCHE
(VAR A: ZAEHLFELD);
```

Die Prozedur LOESCHEN muß also mit einer Variablen vom Typ ZAEHLERFELD aufgerufen werden. Die Variable besitzt innerhalb der Prozedur den Namen A. Beinhaltet eine Prozedur mehrere Parameter, so müssen die aktuellen Parameter in der Reihenfolge der formalen Parameter angegeben werden. Der Aufbau der Parameterliste kommt später genauer zur Sprache.

2. Nach dem Prozedurkopf folgen alle Deklarationen, wie sie auch beim Hauptprogramm möglich sind. Das heißt, daß eine Prozedur wie ein eigenständiges Programm mit eigenen Konstanten, Typen, Variablen und wiederum Prozeduren ausgestattet sein kann. So besitzt zum Beispiel die Prozedur LOESCHEN eine Variable CH vom Typ CHAR, die nur innerhalb der Prozedur gültig ist, und damit insbesondere nicht von Anweisungen im Hauptprogramm oder in anderen Prozeduren verändert werden kann. Die Prozedur STATISTIK enthält selbst eine Prozedurdeklaration (BALKEN).

3. Schließlich folgt zwischen den Schlüsselworten BEGIN und END der Anweisungsteil der Prozedur. Es ist üblich, nach dem letzten END in Kommentarklammern den Namen der Prozedur zu nennen, die an dieser Stelle endet, damit man bei großen Programmen jeden Anweisungsteil leicht dem Prozedurkopf zuordnen kann.

Im Beispielprogramm »Häufigkeit« werden also nacheinander die Anweisungsteile der Prozeduren LOESCHEN, LESEN und STATISTIK ausgeführt, wodurch nacheinander die Zähler Z gelöscht, erhöht und schließlich grafisch dargestellt werden.

Jetzt ist es an der Zeit, daß Sie etwas mehr über die Sichtbarkeitsregeln von Namen in Pascal erfahren. Sie legen fest, welche Namen in einer Prozedur zur Anwendung kommen. Zunächst noch eine Definition: Ein Block ist eine Prozedur oder das Hauptprogramm selbst.

In einem Block sind alle die Namen sichtbar (also gültig), die innerhalb dieses Blockes deklariert wurden. Außerdem noch diejenigen, die in einem »umfassenden« Block deklariert wurden. Wurde ein Name sowohl in einem umgebenden Block als auch im Block selbst deklariert, so ist nur die innere Deklaration sichtbar.

Variablen, die in einer Prozedur festgelegt wurden, heißen lokale Variablen dieser Prozedur. Benutzt jedoch eine Prozedur Variablen, die in einem umgebenden Block deklariert wurden, so nennt man diese global.

Diese Regeln soll das Programm »Sichtbarkeit« in Listing 3 verdeutlichen. Zur Unterstützung ist in Bild 2 die Schachtelung der Blöcke grafisch dargestellt. Die Prozedur P1 besitzt drei lokale Namen, nämlich den Parameter V1, die Konstante K4 und die Variable V 2. Durch die Deklaration des Namens »V1:CHAR« in P1 ist der Name »V1: T1« aus dem Hauptprogramm in P1 nicht sichtbar. Andererseits sind die Namen von P1 nicht in P2, P21 oder dem Hauptprogramm sichtbar.

Prozedur P21 zeigt auch, daß Namen über mehrere Blöcke hinweg sichtbar

```
0 | ###################################################
1 | ##################################
2 | ##################################
3 | ##################
4 | ##################
5 |
6 |
7 | ########################################################
8 | ##################################
9 | ############################################################
: |
: |
```

Bild 1. Ergebnisausdruck des Programms »Häufigkeit« (Ausschnitt)

```
PROGRAM TYPEN (INPUT, OUTPUT);

  CONST ANZAHL_AUFGABEN = 400;

  TYPE GROSSE_ZAHL = REAL;
       KLEINE_ZAHL = INTEGER;
       ZEICHEN     = CHAR;
       ERGEBNIS    = BOOLEAN;
       EINKOMMEN   = ARRAY [1970..1986]
       OF GROSSE_ZAHL;
       ZEILE       = ARRAY [1..80] OF
       CHAR;
       TESTBOGEN   = ARRAY [1..ANZAHL_
       AUFGABEN] OF ERGEBNIS;

VAR  A,B,C                  : GROSSE_ZAHL;
     CH1,CH2                : ZEICHEN;
     UEBERSCHRIFT           : ZEILE;
     DEUTSCHLAND, ENGLAND   : EINKOMMEN;
     MEDIZINERTEST          : TESTBOGEN;

BEGIN
END.
```

Listing 2. Beispiele für Typdeklarationen

sind: So in P21 der Name K2 aus dem Hauptprogramm, da P21 in der Prozedur P2 enthalten ist, die selbst zum Hauptprogramm gehört.

Für die Praxis bedeutet dies, daß man in einer Prozedur Namen ohne Rücksicht auf die »Umgebung« deklarieren kann. Tatsächlich versucht man sogar, nur in Ausnahmefällen auf globale Variablen zuzugreifen und alle Werte als Parameter an die Prozeduren zu übergeben. Damit kann man später die Prozedur in einem völlig anderen Programm verwenden, ohne daß Namenskonflikte auftreten.

Nach diesem etwas theoretischen Abschnitt soll eine Reihe von Beispielen diese Regeln illustrieren und außerdem verschiedene Typen von Parametern vorstellen. Das erste Beispiel (Listing 4) zeigt eine Prozedur, die eine INTEGER-Zahl als eine Hexadezimal-Zahl anzeigt. Zur Anzeige wird die vierstellige Zahl zunächst in zwei Byte zerlegt, deren beide Hexadezimal-Stellen schließlich mit der Prozedur PRINTDIGIT angezeigt werden. Klar zu erkennen ist hier die Schachtelung der drei Prozeduren, wobei jede einen Parameter X vom Typ INTEGER besitzt.

Die Prozedur SWAP in Listing 5 besitzt zwei Parameter des Typs INTEGER. Die Prozedur tauscht den Inhalt dieser beiden Parameter aus.

Listing 4 und 5 zeigen die beiden in Pascal vorhandenen Typen von Parametern. Wird ein formaler Parameter im Prozedurkopf mit dem Schlüsselwort VAR gekennzeichnet, so nennt man ihn einen Variablenparameter. In diesem Fall muß der aktuelle Parameter eine Variable des angegebenen Typs sein. Der Aufruf

```
SWAP(5,6)
```

wäre also nicht zulässig. Innerhalb der Prozedur bewirkt jede Zuweisung an einen Variablenparameter eine Änderung des Wertes der Variablen, die als aktueller Parameter übergeben wurde. Wurde also zum Beispiel die Prozedur SWAP mit den Variablen X und Y aufgerufen, so wird durch die Zuweisung A:=B tatsächlich der Variablen X der Wert der Variablen Y zugewiesen.

Die Parameter der Prozedur PRINTHEX sind hingegen Wertparameter. Beim Aufruf einer solchen Prozedur werden die aktuellen Parameter, die nicht unbedingt Variablen sein müssen, in lokalen Variablen der Prozedur gespeichert. Bei der Ausführung der Prozedur wird nur auf diese Kopie des aktuellen Parameters zugegriffen, so daß insbesondere eine Zuweisung an einen Wertparameter niemals eine Änderung im aufrufenden Block bewirkt.

Ist Ihnen der Unterschied noch nicht völlig klar, so sollten Sie das Schlüsselwort VAR aus dem Prozedurkopf von SWAP löschen und das Programm neu compilieren. Nach der Rückkehr aus der Prozedur werden dann die Werte der Variablen X und Y im Hauptprogramm unverändert sein.

Jeder Parameter wird wie eine Variable mit der Angabe seines Typs nach einem Doppelpunkt deklariert. Dabei ist

```
PROGRAM VARIABLENPARAMETER (INPUT,
  OUTPUT);
  VAR X, Y: INTEGER;

  PROCEDURE SWAP (VAR A, B: INTEGER);
  (* TAUSCHE DEN INHALT VON A UND B *)

    VAR H: INTEGER;

  BEGIN
    H:=A; A:=B; B:=H
  END; (* SWAP *)

BEGIN
  X:= 3; Y:= 4;
  SWAP (X, Y);
  WRITELN (X, Y);
END
```

Listing 5. Ein Beispiel für die Verwendung von Wert-Parametern

```
PROGRAM SICHTBARKEIT (OUTPUT);

  CONST K1 = 3; K2 = 'C'; K3 = '-----------';
  TYPE  T1 = INTEGER; T2 = REAL;
  VAR   V1 : T1;

  PROCEDURE P1 (VAR V1: CHAR);

    CONST K4 = 23.4; .
    VAR   V2 : INTEGER;

  BEGIN
    (* HIER SIND FOLGENDE NAMEN SICHTBAR:
      CONST K1 = 3; K2 = 'C'; K3 = '-----------'; K4= 23.4;
      TYPE  T1 = INTEGER; T2 = REAL;
      VAR   V1 : CHAR;
            V2 : INTEGER;           *)
  END; (* P1 *)

  PROCEDURE P2;
    VAR V2: REAL;

    PROCEDURE P21;
      CONST K1 = '2';

    BEGIN
    (* HIER SIND FOLGENDE NAMEN SICHTBAR:
      CONST K1 = '2'; K2 = 'C'; K3 = '-----------';
      TYPE  T1 = INTEGER; T2 = REAL;
      VAR   V1 : T1;
            V2 : REAL;           *)
    END; (* P21 *)
  BEGIN (* ANFANG DES ANWEISUNGSTEILS VON P1 *)
    (* HIER SIND FOLGENDE NAMEN SICHTBAR:
      CONST K1 = 3; K2 = 'C'; K3 = '-----------';
      TYPE  T1 = INTEGER; T2 = REAL;
      VAR   V1 : T1;
            V2 : REAL;           *)
  END; (* P2 *)

BEGIN
  (* HIER SIND FOLGENDE NAMEN SICHTBAR:
    CONST K1 = 3; K2 = 'C'; K3 = '-----------';
    TYPE  T1 = INTEGER; T2 = REAL;
    VAR   V1 : T1;
          V2 : REAL;           *)
END.
```

Listing 3. Ein Beispiel für Sichtbarkeitsregeln

```
PROGRAM WERTPARAMETER (INPUT, OUTPUT);

  VAR X: INTEGER;

  PROCEDURE PRINTHEX (X: INTEGER);
  (* DRUCKE GANZE ZAHL X ALS HEXADEZIMALZAHL IM FORMAT
  $XXXX *)

  PROCEDURE PRINTBYTE (X: INTEGER);
  (* DRUCKE ZAHL ZWISCHEN 0 UND 255 ALS BYTE IM FORMAT
  XX *)

      PRINTDIGIT (X: INTEGER);
      (* DRUCKE HEXADEZIMALE ZIFFER *)

      BEGIN
        IF X>9 THEN
          WRITE(CHR(X-10+ORD('A')))
        ELSE
          WRITE(X)
      END; (* PRINTDIGIT *)

   BEGIN
     PRINTDIGIT(X DIV 16);
     PRINTDIGIT(X MOD 16);
   END; (* PRINTBYTE *)

  BEGIN
    WRITE('$');
    PRINTBYTE(X DIV 256);
    PRINTBYTE(X MOD 256);
  END; (* PRINTHEX *)

BEGIN (* HAUPTPROGRAMM *)
  WRITELN('Geben Sie positive ganze Zahlen ein:');
  READ(X);
  WHILE X>0 DO
    BEGIN
      WRITELN(' = '); PRINTHEX(X);
      READ(X)
    END;
END.
```

Listing 4. Umrechnung dezimal nach hexadezimal

```
PROGRAM VEKTOROPERATIONEN (INPUT, OUTPUT);

  CONST N = 3; (* LAENGE EINES VEKTORS = ANZAHL DER ELEMENTE
  IM VEKTOR *)

  TYPE  VEKTOR = ARRAY[1..3] OF REAL;

  VAR   X,Y,Z : VEKTOR;
        SKALAR: INTEGER;

  PROCEDURE HOLEN( VAR V: VEKTOR);
  (* VEKTOR MIT N KOMPONENTEN EINLESEN *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO READ(V[I]);
    READLN;
  END; (* HOLEN *)

  PROCEDURE DRUCKEN( V: VEKTOR);
  (* VEKTOR MIT N KOMPONENTEN DRUCKEN *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO WRITE(V[I]:8);
    WRITELN;
  END; (* DRUCKEN *)

  PROCEDURE ADDIEREN (A,B: VEKTOR; VAR C: VEKTOR);
  (* C = A + B. DIE ADDITION ERFOLGT KOMPONENTENWEISE *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO C[I]:= A[I] + B[I]
  END; (* ADDIEREN *)

  PROCEDURE MULTIPLIZIEREN (S: REAL; A: VEKTOR; VAR C: VEKTOR)
  (* C = S * A. JEDE KOMPONENTE WIRD MIT S MULTIPLIZIERT *)

    VAR I: INTEGER;

  BEGIN
    FOR I:= 1 TO N DO C[I]:= S * A[I]
  END; (* MULTIPLIZIEREN *)

BEGIN
  WRITE('X ='); HOLEN(X);
  WRITE('Y ='); HOLEN(Y);
  ADDIEREN(X,Y,Z);

  WRITE('X + Y ='); DRUCKEN(Z);
  WRITE('F ='); READLN(SKALAR);
  MULTIPLIZIEREN (SKALAR,X,Z);
  WRITE('F * X ='); DRUCKEN(Z);
END.
```

Listing 6. Vektoroperationen in Pascal

zu beachten, daß in der Parameterliste nur Typnamen auftreten dürfen. Diese müssen Sie also eventuell zunächst (außerhalb der Prozedur) im Typvereinbarungsteil bestimmt haben. Statt

```
PROCEDURE DRUCKE(X:ARRAY [1..2]
OF CHAR);
```

muß man schreiben

```
TYPE T: ARRAY [1..2] OF CHAR;
PROCEDURE DRUCKE(X: T);
```

Es gibt keinerlei Beschränkung für die Typen der Variablen- oder Wertparameter. Dies soll das Beispiel in Listing 6 verdeutlichen. In der Mathematik würde man ein Array mit N Elementen des Typs REAL als einen Vektor reeller Zahlen bezeichnen. Mit Vektoren kann man wie mit »normalen« Zahlen rechnen. Sind X und Y zwei Vektoren und S eine reelle Zahl, so kann man zum Beispiel X+Y und S*X berechnen. Damit Sie auch mit diesen Operationen ein wenig experimentieren können, sind neben den Prozeduren ADDIEREN und MULTIPLIZIEREN noch die Prozeduren HOLEN und DRUCKEN zur Ein- und Ausgabe von Vektoren vorhanden.

An diesen vier Prozeduren erkennen Sie gut den Unterschied zwischen Wert- und Variablenparametern. Nur wenn über einen Parameter ein Ergebnis oder eine Eingabe zurückgeliefert werden soll, verwendet man Variablenparameter, ansonsten Wertparameter. Dadurch ist während der Ausführung der aktuelle Parameter gegen (unbeabsichtigtes) Überschreiben geschützt.

Es gibt jedoch einen weiteren Fall, in dem man mit Variablenparametern arbeitet, obwohl keine Ergebnisse zurückgeliefert werden sollen. Werden sehr große Variablen an ein Unterprogramm übergeben, so existiert jeder Parameter im Speicher des Rechners doppelt. Einerseits wird der Wert der Variablen im aufrufenden Programm gespeichert und zusätzlich beim Aufruf der Prozedur als lokale Variable. Für die Praxis können Sie sich also merken, daß Sie große Arrays (insbesondere auf Mikrocomputern mit ihrem kleinen adressierbaren Speicherraum) am besten als Variablenparameter übergeben. Bei diesen wird nämlich nur eine Adresse (also nur wenige Bytes) an die Prozedur übergeben, die die Position des aktuellen Parameters im Speicher bezeichnet. In der Prozedur wird dann jeder Zugriff auf diesen Parameter indirekt über die Adresse ausgeführt.

Wegen dieser unterschiedlichen Formen der Übergabe bezeichnet man den Aufruf mit Wertparametern als »call by value« und den Aufruf mit Variablenparametern als »call by reference«.

Zu diesem Themengebiet der »Technik hinter den Kulissen« gehört auch die Verwaltung des Speichers bei einem Pascal-Rechner. Neben den (statistischen) Sichtbarkeitsregeln für die Namen von Variablen, muß auch die (dynamische) Gültigkeit der Werte von Variablen Beachtung finden.

Beim Eintritt in einen Block ist der Wert jeder Variablen, die nicht Parameter einer Prozedur oder Funktion ist, undefiniert.

So ist zum Beispiel am Programmanfang jede Variable unbestimmt. Sie besitzt also nicht etwa wie in Basic den Wert Null. Bei jedem neuen Aufruf einer Prozedur verhält es sich mit allen lokalen Variablen bis auf die Parameter ebenso. Man kann also nicht davon ausgehen, daß die Variablen die Werte ihres letzten Aufrufes beibehalten.

Vielleicht interessiert es Sie, die Hintergründe dieser Regel zu erfahren. Bei der Übersetzung eines Pascal-Programms bestimmt der Compiler für jede Prozedur den Speicherplatzbedarf für alle lokalen Variablen. Innerhalb dieses Speichers weist er jeder Variablen eine feste Position zu. Jedoch bleibt die absolute Lage des Speicherblockes im Computer unbestimmt.

Erst während der Ausführung wird beim Aufruf jeder Prozedur der Speicherblock für die lokalen Variablen reserviert. Umgekehrt wird beim Ende der Ausführung einer Prozedur ihr gesamter Speicherblock wieder freigegeben. Unterprogramme haben aber bekanntlich die Eigenschaft, das zuletzt aufgerufene Unterprogramm als erstes wieder zu verlassen. Ein Beispiel:

```
A ruft B
  B ruft C
    C ruft D
      D kehrt zurück zu C
    C kehrt zurück zu B
  B kehrt zurück zu A
Ende von A
```

Deshalb werden die Variablen eines Pascal-Programms auf einem Stack (Stapelspeicher) verwaltet.

Betrachten wir das Programm in Listing 7. Es besteht aus zwei Prozeduren, die in verschiedener Reihenfolge aufgerufen werden (ansonsten aber nicht viel Sinnvolles erledigen). Wir wollen nun nach jedem Schritt einen Blick auf den Speicher des Rechners werfen (Bild 3):

Zu Programmbeginn (1) ist der gesamte Speicher frei. Beim Eintritt in das Hauptprogramm wird zunächst Platz für die Variable M des Hauptprogramms geschaffen (2). Nun erfolgt der Aufruf der Prozedur P1, die ihrerseits Platz für die Variable L1 benötigt (3). Nach der Rückkehr aus P1 kann dieser Platz sofort wieder freigegeben werden (4). Der Aufruf P2(FALSE) erfolgt in denselben Schritten (5 und 6), wobei jedoch Platz für zwei Variablen (L2 und auch P1) benötigt wird.

Sie sehen schon jetzt, daß derselbe Speicherplatz sowohl für die Variablen von P1 als auch von P2 verwendet wird. Beim Aufruf von P2(TRUE) (7), ergibt sich zunächst der Zustand von (5), jedoch wird außerdem In P2 noch P1 aufgerufen, so daß sich schließlich eine Speicherverteilung wie in (8) ergibt. Offensichtlich liegt beim zweiten Aufruf von P1 die Variable L2 an einer anderen absoluten Adresse. Bei der Rückkehr aus P1(9) und P2(10) werden wieder die lokalen Variablenbereiche freigegeben.

Der Stack »wächst« also von unten nach oben und nimmt von dort wieder nach unten ab, wobei er immer einen zusammenhängenden Speicherbereich bildet. Daß Ihr Rechner zur Laufzeit der Programme einen Stack verwaltet, merken Sie spätestens dann, falls bei einem Prozeduraufruf kein Platz mehr für die lokalen Variablen vorhanden ist. Das quittiert das Programm gewöhnlich mit der Fehlermeldung »stack overflow«.

Funktionen

Inzwischen sind Sie mehrmals auf die Formulierung »Prozedur« oder »Funktion« gestoßen, ohne daß Sie Näheres über Funktionen in Pascal erfahren haben. Eine Funktion ist eine spezielle Form einer Prozedur, die zusätzlich noch einen Wert als Ergebnis liefert. In der Mathematik gibt es zum Beispiel die Maximumfunktion, die das Maximum von zwei Zahlen liefert, so daß gilt:

$$\max\{3,4\} = 4$$
$$3 + \max\{0,-7\} = 3$$

Man kann also das Ergebnis der Funktion auch in arithmetischen Ausdrücken verwenden. All diese Möglichkeiten gelten bei der Verwendung von Funktionen in Pascal, die einige Standardfunktionen der Mathematik nachbilden. Dabei verbirgt sich unter der Funktion HOCH (Listing 8) die in Pascal standardmäßig nicht vorhandene Methode, um A hoch B für beliebige Zahlen zu berechnen.

Bei der Definition unterscheidet sich eine Funktion von einer Prozedur nur durch den Funktionskopf. Hier ersetzt das Schlüsselwort FUNCTION das Wortsymbol PROCEDURE. Außerdem wird zusätzlich am Ende der (eventuell leeren) Parameterliste nach einem Doppelpunkt der Name des Typs angegeben, zu dem das Ergebnis der Funktion gehört. Somit lautet der Funktionskopf der Funktion MAX, die das (reelle) Maximum zweier reeller Zahlen berechnet, folgendermaßen:

```
FUNCTION MAX (A,B: REAL): REAL;
```

Der Ergebnistyp darf kein zusammengesetzer Typ, wie zum Beispiel eine Menge oder ein Array, sein. Um innerhalb der Funktion das Ergebnis zu bestimmen, verwendet man den Funktionsnamen in einer Zuweisung:

```
IF A>B THEN
   MAX:=A
ELSE
MAX:=B
```

Natürlich muß auf jeden Fall innerhalb einer Funktion eine solche Zuweisung stattfinden, damit die Funktion bei der Rückkehr einen definierten Wert liefert. Funktionsaufrufe sind nur innerhalb von Ausdrücken zulässig, während Prozeduraufrufe syntaktisch gesehen Anweisungen sind.

```
IF MAX(A,B) >4 THEN ...
A := MAX(A,B);
MIN:= -MAX(-A,-B)
```

Jetzt folgt eine kleine Sammlung von Prozeduren und Funktionen, die Ihnen die Programmierung typischer Operationen mit Arrays zeigt. Die erste Prozedur in Listing 9 sortiert den Inhalt des Arrays in A, das als Parameter übergeben wird. Durch die Wahl von SORTTYPE = INTEGER könnten Sie diese Prozedur auch zum Sortieren ganzer Zahlen (oder jedes anderen Typs) verwenden. Das Sortierverfahren ist eines der einfachsten und langsamsten überhaupt. Bei jedem Durchlauf der äußeren FOR-Schleife wird jeweils ein Wert im Array an seine korrekte Position gebracht und dazu in der inneren Schleife der maximale Wert im Restarray bestimmt. Somit wird im ersten Durchlauf die größte Zahl mit der Zahl an der letzten Arrayposition vertauscht. Im nächsten Durchlauf wechselt die zweitgrößte Zahl mit der vorletzten Arrayposition den Platz, bis im letzten Durchlauf der kleinste Wert an der ersten Position landet.

Eine etwas exotischere Prozedur stellt Listing 10 vor. Hier wird der Kehrwert des ganzzahligen Parameters I exakt gedruckt. Bei periodischen Brüchen erscheint ein Pfeil am Beginn der Periode.

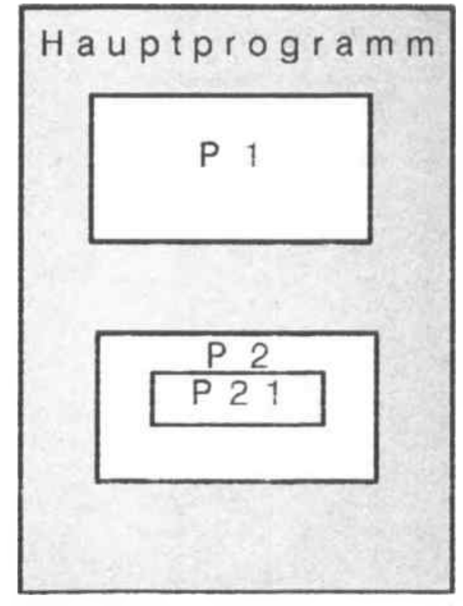
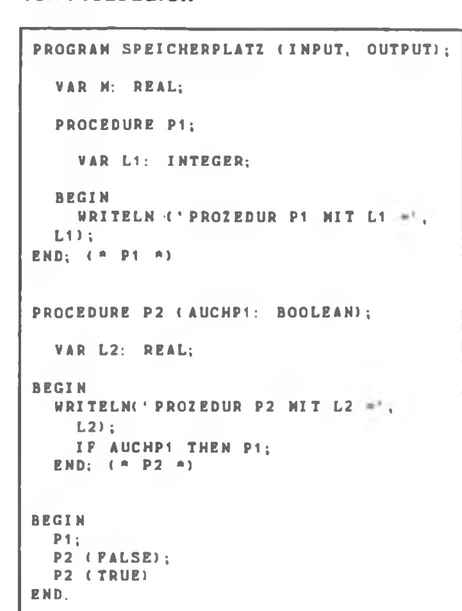


Bild 2. Statische Schachtelung von Prozeduren



```
PROGRAM SPEICHERPLATZ (INPUT, OUTPUT);

  VAR M: REAL;

  PROCEDURE P1;

    VAR L1: INTEGER;

  BEGIN
    WRITELN ('PROZEDUR P1 MIT L1 =',
  L1);
END; (* P1 *)


PROCEDURE P2 (AUCHP1: BOOLEAN);

  VAR L2: REAL;

BEGIN
  WRITELN('PROZEDUR P2 MIT L2 =',
    L2);
    IF AUCHP1 THEN P1;
  END; (* P2 *)


BEGIN
  P1;
  P2 (FALSE);
  P2 (TRUE)
END.
```

Listing 7. Beispielprogramm zur Speicherorganisation (vergleiche Bild 3).

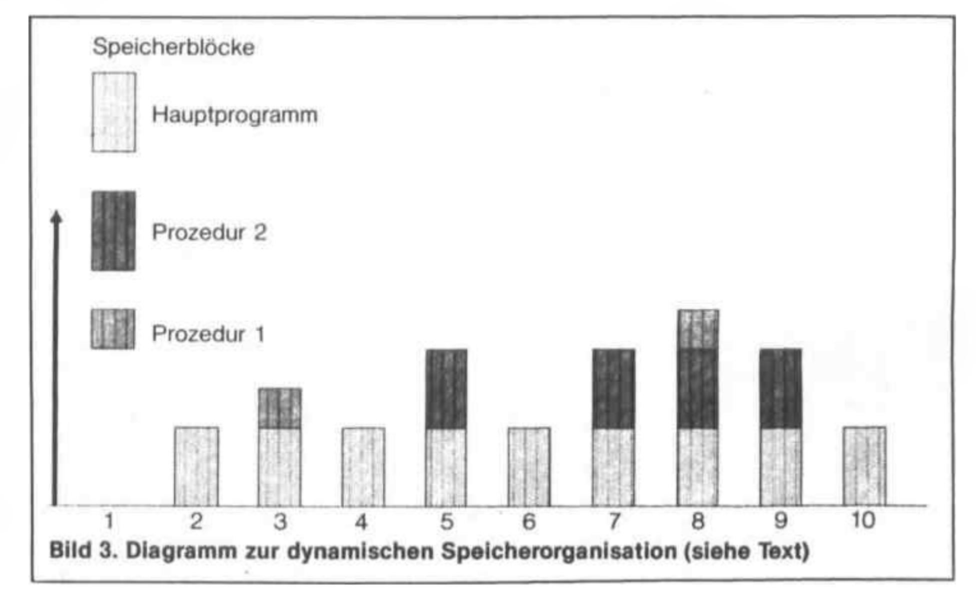


Bild 3. Diagramm zur dynamischen Speicherorganisation (siehe Text)

```
1/3  = 0. 13
1/12 = 0. 08^3
1/17 = 0.10588235294117647
```

Wie findet man aber den Anfang (und damit auch das Ende) der Periode? Zunächst erinnern wir uns an die Rechnung mit Brüchen aus der Schule:

```
1. 000000 / 7 = 0. 1142857
1 0
  30
   20
     60
       40
         50
           10
```

Beginnend mit dem REST 1 multipliziert man immer 10 und dividiert das Produkt wieder durch I. Der ganzzahlige Anteil REST DIV I ist die nächste Ziffer des Ergebnisses, während der Divisionsrest REST MOD I den REST für den nächsten Divisionsschritt bildet.

Die Periode endet genau dann, wenn der REST bereits früher einmal in der Berechnung (nicht unbedingt als erster, siehe 1/12!) aufgetreten ist. Daher speichert die Prozedur alle Ziffern in einem Array ZIFFERN und für jeden Rest R den Index der zugehörigen Ziffern in dem Array INDEX. Wird dabei festgestellt, daß dieser INDEX ungleich 0 ist, also dieser Rest bereits einmal auftrat, so ist die gesamte Periode bekannt, und das Ergebnis kann anschließend gedruckt werden. Probieren Sie es doch einmal mit 1/29!

Zum Verständnis der Prozedur Listing 11 benötigen Sie noch Kenntnisse über die Stringbehandlung in Pascal. Bisher wurden in allen Beispielprogrammen nur einzelne Zeichen, also Werte vom Typ CHAR, verwendet. Möchte man jedoch Worte, Zeilen oder Sätze, die aus einer Folge von Zeichen bestehen, verarbeiten, so muß man ein Array aus Zeichen definieren.

```
CONST TEXT='Otto Anna';
TYPE  STRING = ARRAY [1..9]
OF CHAR;
VAR   WORT : STRING;
      ZEILE: ARRAY [1.80]
OF CHAR;
```

Text ist eine Konstante mit dem Typ ARRAY [1..9] OF CHAR, da sie aus neun Zeichen besteht. Denselben Typ besitzt die Variable Wort. Die Variable Zeile umfaßt 80 einzelne Zeichen. Wichtig ist jetzt die Tatsache, daß jedem String eine feste Länge zugeordnet ist. Es ist also nicht möglich, in Wort nur vier Zeichen zu speichern, man muß vielmehr den Rest des Strings zum Beispiel mit Leerzeichen füllen:

```
WORT:= 'ANNA_____';  (korrekt)
WORT:= 'ANNA';       (falsch)
WORT:= TEXT;         (korrekt)
WORT:= ZEILE;        (falsch)
Zeile:= WORT;        (ebenfalls
                     falsch!)
```

```
FUNCTION SIGNUM (X: REAL): INTEGER;
(* Vorzeichen: 0, -1 oder +1 *)

BEGIN
  IF X>0 THEN SIGNUM:= 1
  ELSE
    IF X<0 THEN SIGNUM:= -1
    ELSE SIGNUM:= 0
END; (* SIGNUM *)

FUNCTION HOCH (X,Y: REAL):REAL;
(* ERGEBNIS IST X ^ Y *)
BEGIN
  HOCH:= EXP( Y * LOG(X));
END; (* HOCH *)

FUNCTION FAKULTAET (N:INTEGER):REAL;
(* ERGEBNIS IST N ! = 1 * 2 * 3 * 4 *
... * N *)
            VAR P: REAL;
                I: INTEGER;
BEGIN
  P:= 1.0;
  FOR I:= 2 TO N DO P:= P * I
  FAKULTAET:= P
END; (* FAKULTAET *)

GEOMETRISCHES_MITTEL(A,B:REAL): REAL;
BEGIN
  GEOMETRISCHES_MITTEL:= SQRT(A*B)
END;

ARITHMETISCHES_MITTEL(A,B,C:REAL):
REAL; BEGIN
  ARITHMETISCHES_MITTEL:=
  (A + B + C) / 3
  END;
```

Listing 8. Einige Beispiele für Funktionen in Pascal

Zuweisungen sind also nur zwischen Strings gleicher Länge möglich, da weder überflüssige Zeichen abgeschnitten noch zu kurze Strings mit Leerzeichen erweitert werden. Andererseits kann man jedoch auch Strings gleicher Länge miteinander vergleichen. Dabei bestimmt das Ergebnis des Vergleiches den zugrundeliegenden Zeichensatz. Im ASCII-Code gilt beispielsweise

```
'ALPHA' < 'BETA '
'ALPHA' < 'alpha'
'ALP  ' < 'ALPHA'
```

Die hier gemachten Einschränkungen gelten für Standard-Pascal. Für Turbo-Pascal und andere Compiler gelten sie nicht.

Die Funktion POSITION in Listing 11 liefert die Position von Wort in der Tabelle TAB. Diese Tabelle soll aufsteigend mit Strings gefüllt sein. Aus der Funktion resultiert der Wert 0, falls Wort nicht in TAB enthalten ist.

```
CONST N = 100;

TYPE  SORTIERELEMENT = REAL;
      SORTIERFELD    = ARRAY (UG..OG)
      OF SORTIERELEMENT;

PROCEDURE SORT(VAR A: SORTIERFELD);

  VAR I,J,K: INTEGER;
      MAX  : SORTIERELEMENT;

BEGIN
  FOR J:= OG DOWNTO UG+1 DO
    BEGIN
      MAX:= A(UG]; K:= UG;
      FOR I:= UG+1 TO J DO
        IF A(I] > MAX THEN
          BEGIN
            K:= I; MAX:= A(I)
          END;
      A(K]:= A(J]; A(J]:= MAX
    END;
END; (* SORT *)
```

Listing 9. Sortieren durch Auswahl

Falls Sie einmal ein Programm schreiben möchten, das Ihnen alle Namen in einem Pascal-Programm ausdruckt, so müssen Sie zunächst alle Schlüsselworte erkennen. Hierzu könnten Sie die Funktion POSITION verwenden. In TAB werden alle Schlüsselworte in alphabetischer Reihenfolge eingetragen (AND, ARRAY, BEGIN, ...). Dann ergeben sich folgende Funktionsergebnisse:

```
POSITION('AND       ',TAB) = 1
POSITION('BEGIN     ',TAB) = 3
POSITION('WITH      ',TAB) = 39
POSITION('WIRSING   ',TAB) = 0
```

Obwohl die Funktion etwas komplizierter aussieht, arbeitet sie für große Arrays wesentlich schneller als die einfache Version aus Listing 12, die das Array schrittweise von hinten nach vorne durchsucht. Statt dessen verwendet die erste Funktion zwei Zeiger R und L auf den rechten und linken Rand des Teilarrays, in dem sich der gesuchte Wert befinden muß. In jedem Schritt wird nun der Mittelpunkt M des Intervalls untersucht. Steht an dieser Stelle ein Wort, das größer als das Suchwort ist, so liegt das Suchwort im Intervall L bis M-1, ansonsten muß es im Interval M+1 bis R zu finden sein. Die Suche ist beendet, falls die Zeiger L und R sich überschneiden. Offensichtlich wird in jedem Schritt das Array halbiert, was den Geschwindigkeitsvorteil gegenüber Listing 12 ausmacht.

Noch ein Wort zu der linearen Suche in Listing 12: Warum wurde nicht einfach die Schleife

```
I:= ANZAHL;
WHILE (WORT <>TAB[I] AND
(I <>0) DO I :=I-1;
POSITION:= I
```

verwendet? Ist in diesem Fall Wort nicht in TAB vorhanden, so ist im letzten Schleifendurchlauf I=0, was im ersten Teil der WHILE-Bedingung einen Zugriff auf das (nicht existierende) Element TAB[0] zur Folge hat. Normalerweise erkennt ein Pascal-Laufzeitsystem solche Indizierungsfehler und gibt dann eine Fehlermeldung aus.

Rekursion

Das letzte Programm in Listing 13 ist ein besonders schönes Beispiel für eine sehr nützliche Eigenschaft von Prozeduren und Funktionen in Pascal. Falls Sie sich an die Ausführungen über die Speicherverwaltung erinnern, wissen Sie noch, daß bei jedem Prozeduraufruf neuer Speicherplatz für die lokalen Variablen bereitgestellt wird. Des-

halb kann sich eine Prozedur auch selbst aufrufen, ohne daß dadurch der Inhalt der lokalen Variablen zerstört würde. Den Selbstaufruf einer Prozedur oder Funktion nennt man Rekursion. Das Programm in Listing 13 verwendet nun eine rekursive Prozedur PROBIERE_ZEILE, um das »Problem der acht Damen« zu lösen.

Die Aufgabe besteht darin, acht Königinnen, die nach den Schachregeln alle Figuren in ihrer Zeile und Spalte sowie in beiden Diagonalen bedrohen, so auf ein Schachbrett zu setzen, daß keine Figur eine andere bedroht. Bild 4 zeigt die Felder, die eine Dame angreift, während Bild 5 eine zulässige Lösung zeigt.

Die Lösung stellt ein sogenannter Backtracking Algorithmus. Hierbei probiert man eine Reihe von Schritten aus, bis man feststellt, daß dieser Weg in eine Sackgasse führt. In diesem Moment beginnt man die letzte Entscheidung rückgängig zu machen und einen neuen Weg zur Lösung zu finden. Indem man systematisch alle möglichen Kombinationen durchsucht, wird auf jeden Fall eine Lösung (sofern vorhanden) gefunden.

In diesem konkreten Beispiel positioniert man zunächst eine Dame an eine zulässige Stelle der ersten Zeile. Dabei wird notiert, welche Diagonalen und welche Spalte die Dame bedroht. Anschließend wird in der zweiten Reihe eine ungefährdete Position gesucht, die die zweite Dame besetzt. Diese Prozedur wiederholt sich so lange, bis die achte Zeile belegt und die Lösung gefunden ist. Im Normalfall ist natürlich bereits in der dritten oder vierten Zeile jedes Feld gefährdet, so daß die Suche in eine Sackgasse führt. Dann wird die Dame von ihrem Feld wieder entfernt und in der letzten Reihe die Dame auf das nächste freie Feld gesetzt. Mit dieser Strategie erhält man alle 92 Lösungen auf einem 8 x 8-Brett.

Das größte Problem besteht darin, möglichst schnell zu testen, ob eine Diagonale oder eine Spalte bereits belegt ist. Deshalb werden drei boolesche Arrays geführt, die für jede Diagonale und Spalte den Wert TRUE enthalten, falls diese nicht bedroht ist. Natürlich muß die Information in diesem Array am Programmanfang gelöscht und bei jedem Zug und jeder Zugrücknahme aktualisiert werden. Bild 6 zeigt, wie man aus dem Zeilen- und Spaltenindex I,J die Nummer der jeweiligen Diagonalen durch Addition und Subtraktion erhält.

Bisher wurde nur das Array als zusammengesetzter Typ verwendet. Durch die Flexibilität bei der Wahl des Indextyps und des Typs der Elemente konnten sehr verschiedenartige Probleme behandelt werden. Bevor wir uns

```
BEGIN
  IF I>MAXIMAL THEN  WRITELN('DIE ZAHL IST ZU GROSS!')
  ELSE
    BEGIN
      WRITE('1 / ', I, ' = 0.');
      (* NOCH IST KEIN REST AUFGETRETEN, DESHALB INDEX LOESCHEN: *)
      FOR J:= 0 TO I-1 DO
        INDEX[J]:= 0;

      K    := 0;         (* DIE ERSTE ZIFFER WIRD BERECHNET *)
      REST:= 1;          (* WIR BERECHNEN   1 / I           *)
      REPEAT
        K:= K+1; INDEX[REST]:= K;
        REST:= ZEHN * REST;
        ZIFFER[K]:= REST DIV I;
        REST     := REST MOD I;
      UNTIL INDEX[REST]<>0;       (* BIS DIESER REST SCHON EINMAL BERECHNET *)

      (* JETZT NOCH DAS ERGEBNIS DRUCKEN, INDEX[REM] IST DIE POSITION DER  *)
      (* ERSTEN ZIFFER IN DER PERIODE                                      *)
      FOR J:= 1 TO INDEX[REST]-1 DO
         WRITE(CHR(ZIFFER[J]+ORD('0')));
      WRITE([illegible]);
      FOR J:=INDEX[REST] TO K DO
         WRITE(CHR(ZIFFER[J]+ORD('0')));
      WRITELN
    END;
END; (* KEHRWERT *)
PROCEDURE KEHRWERT (I: INTEGER);
(* DRUCKE DEN EXAKTEN KEHRWERT DER ZAHL I MIT ANGABE DER PERIODE        *)

   CONST ZEHN = 10;          (* BASIS DES DEZIMALSYSTEMS                *)
         MAXIMAL = 300;      (* MAXIMALE GROESSE FUER I UND DIVISIONSREST *)

   VAR   J,K       : INTEGER;
         REST      : INTEGER;(* LAUFENDER DIVISIONSREST BEI STELLE K    *)

         ZIFFER    : ARRAY[1..MAXIMAL] OF INTEGER;
         INDEX     : ARRAY[0..MAXIMAL] OF INTEGER;
```

Listing 10. Exakte Berechnung des Kehrwertes

```
CONST ANZAHL = 30;                        (* LAENGE DER SUCH-TABELLE       *)

TYPE  STRING = ARRAY[1..11] OF CHAR; (* LÄNGE EINES WORTES = 11 ZEICHEN *)
      TABELLE= ARRAY[1..ANZAHL] OF STRING;

FUNCTION POSITION (WORT: STRING; VAR TAB: TABELLE);
   TAB MUSS AUFSTEIGEND ALPHABETISCH SORTIERT SEIN. DAS ERGEBNIS DER    *)
   FUNKTION IST DIE POSITION VON WORT IN TAB BZW. 0, FALLS WORT NICHT   *)
(* IN TAB STEHT.                                                        *)

  VAR L,R,M: INTEGER;

BEGIN
  L:=1; R:=ANZAHL;
  REPEAT
    M:= (L+R) DIV 2;                      (* MITTELPUNKT DES INTERVALLS L..R *)
    IF WORT<=TAB[M] THEN R:= M-1;         (* WORT LIEGT LINKS VON DER MITTE  *)
    IF WORT>=TAB[M] THEN L:= M+1;         (* WORT LIEGT RECHTS VON DER MITTE *)
  UNTIL L>R;                              (* BIS ZEIGERKOLLISION             *)
  IF L>R+1 THEN POSITION:=R+1;            (* WORT GEFUNDEN                   *)
           ELSE POSITION:=0               (* WORT NICHT VORHANDEN            *)
END; (* POSITION *)
```

Listing 11. Binäre Suche

```
PROCEDURE POSITION (WORT: STRING; VAR TAB: TABELLE);
(* EINFACHE LINEARE SUCHE NACH WORT IN TAB. TAB MUSS NICHT SORTIERT SEIN *)

  VAR I: INTEGER;

BEGIN
  I:=ANZAHL;
  WHILE (TAB[I]<>WORT) AND (I<>1) DO I:=I-1;
  IF TAB[I]=WORT THEN POSITION:=I
                 ELSE POSITION:=0
END; (* POSITION *)
```

Listing 12. Lineare Suche

weiteren Datentypen in Pascal zuwenden, kommen noch zwei weitere Eigenschaften von Arrays zur Sprache.

1. Man kann auch das Array als Ganzes in einer Zuweisung verwenden, falls auf der rechten und linken Seite des Zuweisungsoperators je ein Array desselben Typs steht:

```
VAR A,B: ARRAY[1..100] OF REAL;
A:=B;
```

Mit dieser Zuweisung werden also 100 reelle Zahlen aus dem Array B in das Array A kopiert. Somit enthält diese allgemeine Regel die oben genannten Bedingungen, unter denen eine Zuweisung zwischen Strings zulässig ist.

2. Arrays können als Elemente nicht nur einfache, sondern auch zusammengesetzte Typen enthalten. Wiederum hatten wir diese Tatsache bereits bei der Besprechung der Strings vorweggenommen, als wir definierten:

```
TYPE STRING = ARRAY[1..11]
OF CHAR;
     TABELLE= ARRAY[1..ANZAHL]
OF STRING;
```

Ist der Elementtyp ein Array, so spricht man oft auch von mehrdimensionalen Arrays. In diesen Fällen kann man auch eine vereinfachte Notation verwenden. Statt

```
VAR A,B: ARRAY [1..N] OF
           ARRAY [1..M] OF
             ARRAY [1..K] OF REAL;
A[1][1][1]:= B[2][3][4]
```

schreibt man dann:

```
VAR A,B: ARRAY[1..N,1..M,1..K]
OF REAL;
A[1,1]:=B[2,3,4]
```

Bei der Arbeit mit solchen mehrdimensionalen Tabellen verwendet man sehr häufig Ausschnittstypen. Betrachten Sie zum Beispiel (Bild 7), das eine zweidimensionale Tabelle mit Umsatzzahlen (Millionen DM ?) für fünf Jahre mit jeweils zwölf Monaten enthält. Das kann man schreiben als:

```
TYPE MONAT = 1..12;
     JAHR  = 1980..1985;
VAR UMSATZTABELLE = ARRAY
[JAHR,MONAT]
                          OF REAL;
    M : MONAT;
    J : JAHR;
```

Die Variablen M und J besitzen die Ausschnittstypen MONAT und JAHR und können daher nur die Werte 1 bis 12 oder 1980 bis 1985 annehmen. Sie eignen sich also hervorragend als Zeiger in die zweidimensionale Tabelle. Jede Zuweisung an eine Variable eines Ausschnittstyps wird zur Laufzeit auf gültige Werte geprüft, so daß bei folgenden Anweisungen das Programm mit einer Fehlermeldung endet:

```
J:= 12; M:= 1985
```

Einmal schützt also der Compiler vor irrtümlichen Zuweisungen. Listing 14 zeigt drei Funktionen, die in solchen Umsatztabellen Zeilen-, und Spalten-, oder Gesamtsummen berechnen. Das Hauptprogramm verwendet diese Funktionen für die Ermittlung von Umsätzen für zwei Produkte (Farben und Lacke).

Allgemein kann man von jedem skalaren Typ Ausschnittstypen bilden:

```
TYPE BUCHSTABEN = 'A'..'Z';
     ZIFFERN    = '0'..'9';
```

Eine Variable eines Ausschnittstyps kann überall dort verwendet werden, wo auch eine Variable des zugehörigen Grundtyps (INTEGER oder CHAR) stehen kann. Unter Umständen kann die Verwendung von Ausschnittstypen (zum Beispiel als Elementtyp in großen Arrays) bei einigen Compilern auch Speicherplatz einsparen. Mit der Kenntnis, daß die Variable JAHR nur die Werte 1980 bis 1985 annnimmt, genügt 1 Byte zur Speicherung des Wertes (statt 2 oder 4 Byte für die normalen INTEGER-Zahlen).

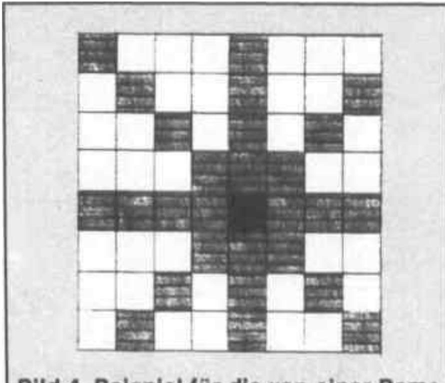


Bild 4. Beispiel für die von einer Dame bedrohten Felder



Bild 5. Eine von mehreren Lösungen des Acht-Damen-Problems

	1	2	3	4	5	6	7	8
1	2	3	4	5	6	7	8	9
2	3	4	5	6	7	8	9	10
3	4	5	6	7	8	9	10	11
4	5	6	7	8	9	10	11	12
5	6	7	8	9	10	11	12	13
6	7	8	9	10	11	12	13	14
7	8	9	10	11	12	13	14	15
8	9	10	11	12	13	14	15	16

	1	2	3	4	5	6	7	8
1	0	1	2	3	4	5	6	7
2	-1	0	1	2	3	4	5	6
3	-2	-1	0	1	2	3	4	5
4	-3	-2	-1	0	1	2	3	4
5	-4	-3	-2	-1	0	1	2	3
6	-5	-4	-3	-2	-1	0	1	2
7	-6	-5	-4	-3	-2	-1	0	1
8	-7	-6	-5	-4	-3	-2	-1	0

Bild 6. Berechnung der Diagonalen im Programm ACHT DAMEN (links: Addition der Zeilen- und Spaltennummern, rechts entsprechend Subtraktion).

Eigene Datentypen definieren

Eine andere Methode, neue Typen zu definieren, besteht darin, daß man alle Werte aufzählt, die dieser Typ annimmt:

```
TYPE AMPEL = (ROT, GELB, GRUEN);
     FAMILIENSTAND = (LEDIG,
                      VERHEIRATET,
                      GETRENNT,
                      GESCHIEDEN,
                      VERWITWET);
     FIGUR         = (BAUER,
                      LAEUFER,
                      SPRINGER,
                      TURM,
                      DAME,
                      KOENIG);
     FARBE         = (SCHWARZ,
                      WEISS);
```

Diese Typen nennt man Aufzählungstypen. Formal gesehen sind zum Beispiel ROT und KOENIG Konstanten des jeweiligen Typs. Durch die Reihenfolge bei der Typendeklaration wird eine Ordnung auf den Konstanten definiert:

```
ORD(ROT) = 0
ORD(GELB) = 1
ORD(GRUEN) = 2
```

Somit sind auch Vergleiche zwischen Variablen eines Aufzählungstyps sinnvoll:

```
VAR HAUPTAMPEL : AMPEL;
IF HAUPTAMPEL <= GELB THEN ...
```

Um den Nachfolger und Vorgänger im Wertebereich zu erhalten, gibt es die Funktionen PRED (predecessor) und SUCC (successor):

```
SUCC(ROT) = GELB
SUCC(LAEUFER) = SPRINGER
PRED(VERHEIRATET) = LEDIG
PRED(WEISS) = SCHWARZ
```

ORD, SUCC und PRED sind übrigens für jeden skalaren Typ zulässig, da alle skalaren Typen geordnet sind:

```
ORD(FALSE)=0
SUCC(FALSE)=TRUE
SUCC('A') = 'B'
PRED(0) = -1
```

Viele hoffnungsvolle Programme von Anfängern enthalten direkte Ein- und Ausgabeanweisungen für Werte von Aufzählungstypen:

```
WRITE (HAUPTAMPEL);
READ (HAUPTAMPEL);
```

Diese Anweisungen sind falsch! Nur während der Übersetzung sind dem Compiler die Namen der Werte des Typs bekannt. Zum Zeitpunkt der Ausführung sind alle Aufzählungstypen durch Zahlen ohne Verweise auf irgendwelche Namen codiert. Man muß also die Umwandlung zwischen dem Aufzählungstyp und dem auszugebenden Namen selbst vornehmen:

```
PROCEDURE DRUCKE_AMPEL(A:AMPEL);
BEGIN
   WRITE('Die Ampel zeigt);
   CASE A OF
    ROT  :  WRITELN('rot');
    GELB :  WRITELN('gelb');
    GRUEN:  WRITELN('grün');
  END;
END; (* DRUCKE_AMPEL *)
```

Natürlich können auch Ausschnittstypen von Aufzählungstypen gebildet werden:

```
TYPE ENDSPIELFIGUR =
LAEUFER..KOENIG;
```

Mit den Aufzählungstypen lernen Sie den letzten skalaren Typen in Pascal kennen. Alle weiteren Typen, die den Rest des Artikels einnehmen, sind zusammengesetzte Typen wie das Array. Mengen von Zahlen sind Ihnen sicher bekannt. Um einen Mengentyp zu deklarieren, verwendet man die Wortsymbole SET und OF, denen ein skalarer Typ folgen muß:

```
TYPE FIGUREN = SET OF FIGUR;
VAR MEINE, DEINE: FIGUREN;
```

Mit den Mengen-Variablen MEINE und DEINE kann man beispielsweise über die geschlagenen Figuren führen. Am Spielbeginn gilt:

```
MEINE:= [BAUER...KOENIG];
DEINE:= MEINE;
IF [TURM,DAME,KOENIG]<
=MEINE THEN...
```

Mit dieser Abfrage wird also getestet, ob ein Spieler noch Turm, Dame und König besitzt. Da eine Menge jedes Element nur einmal oder keinmal enthält, werden Mengen überall dort verwendet, wo das Vorhandensein einer gewissen Eigenschaft signalisiert werden soll.

Arrays und Mengen fassen mehrere Werte eines Typs zu einem zusammengesetzen Typ zusammen. Besonders in großen Programmen ist es jedoch auch sinnvoll, Werte verschiedener Typen zu einer Einheit zu verbinden. Zur Konstruktion solcher Typen verwendet man in Pascal Recordtypen (Verbundtypen):

```
TYPE POSITION = RECORD
                  X,Y: REAL;
                END;
 ADRESSE  = RECORD
      NAME, VORNAME: STRING;
      ORT, STRASSE : STRING;
      HAUSNUMMER   : INTEGER;
      PLZ          : INTEGER;
  END;
```

```
PROGRAM ACHT_DAMEN (INPUT,OUTPUT);

CONST N      = 8;            (* ACHT DAMEN AUF EINEM 8 * 8 SCHACHBRETT *)
      N2     = 16;           (* = 2 * N *)

"\R   DAME        : ARRAY [1..N]  OF INTEGER;
      SPALTE_FREI: ARRAY [1..N]  OF BOOLEAN;
      DIAG1_FREI : ARRAY [2..N2] OF BOOLEAN;
      DIAG2_FREI : ARRAY [-N. N] OF BOOLEAN;
      LOESUNG    : INTEGER;
      I          : INTEGER;

  PROCEDURE DRUCKELOESUNG;
  (* ZEIGE DIE POSITION DER DAMEN AN. ERHÖHE DEN ZÄHLER FÜR DIE LÖSUNGEN *)

    VAR ZEILE, SPALTE: INTEGER;

  BEGIN
    LOESUNG:= LOESUNG + 1;
    WRITELN; WRITELN; WRITELN('LÖSUNG', LOESUNG:2, ':');
    FOR ZEILE:=1 TO N DO
      BEGIN
        FOR SPALTE:=1 TO N DO
          IF SPALTE = DAME[ZEILE] THEN WRITE('*')
                                  ELSE WRITE('+');
        WRITELN;
      END;
  END; (* DRUCKELOESUNG *)

  PROCEDURE SETZE_ZEILE(I: INTEGER);
  (* SUCHE FREIE POSITION FÜR DAME IN ZEILE I. FALLS EIN PLATZ GEFUNDEN *)
  (* WURDE, WIRD EINE DAME IN ZEILE I+1 GESETZT ODER DIE VOLLSTAENDIGE  *)
  (* STELLUNG ANGEZEIGT. SONST ERFOLGT EIN RUECKSPRUNG.                 *)

    VAR J: INTEGER;

  BEGIN
    FOR J:= 1 TO N DO
      IF SPALTE_FREI[J] AND DIAG1_FREI[I+J] AND DIAG2_FREI[J-I] THEN
        BEGIN (* HIER KANN JETZT DIE DAME GESETZT WERDEN: *)
          DAME[I]:= J;
          SPALTE_FREI [J] := FALSE;
          DIAG1_FREI [I+J]:= FALSE;
          DIAG2_FREI [J-I]:= FALSE;

          IF I=N THEN (* ALLE DAMEN GESETZT, LÖSUNG DRUCKEN! *)
            DRUCKELOESUNG
          ELSE        (* IN DER NÄCHSTEN ZEILE WEITERSUCHEN  *)
            SETZE_ZEILE(I+1);

          (* JETZT DEN LETZEN ZUG RÜCKGÄNGIG MACHEN:         *)
          SPALTE_FREI [J] := TRUE;
          DIAG1_FREI [I+J]:= TRUE;
          DIAG2_FREI [J-I]:= TRUE;
        END;
  END; (* SETZE_ZEILE *)

BEGIN
  (* SPIELBRETT IST NOCH LEER: *)
  FOR I:= 1 TO 8  DO SPALTE_FREI (I):= TRUE;
  FOR I:= 2 TO N2 DO DIAG1_FREI  (I):= TRUE;
  FOR I:=-N TO N  DO DIAG2_FREI  (I):= TRUE;
  LOESUNG:= 0;
  SETZE_ZEILE(1);
END.
```

Listing 13. Das Acht-Damen-Problem, ein Beispiel für einen rekursiven Algorithmus.

	Monat											
	1	2	3	4	5	6	7	8	9	10	11	12
1980	1.1	2.1	3.4	5.5	6.6	0.4	3.1	0.0	8.2	9.9	0.0	3.2
1981	1.2	2.3	3.4	4 9	6.2	1.4	3.2	0.1	7.9	9.5	0.2	3.3
1982	1.1	1.9	3.2	4.5	6.3	1.7	3.4	0.2	8.0	9.5	0.0	3.1
1983	1.4	2.2	3.3	4.3	6.6	2.3	3.5	0.3	8.1	9.5	0.2	3.0
1985	1.8	2.3	3.4	4.1	6.7	2.4	3.6	0.4	8.3	9.3	0.0	3.5

Bild 7. Darstellung von Jahresumsätzen als Beispiel

```
VAR P1,P2 : POSITION;
    ADR   : ADRESSE;
```

Nach einer solchen Deklaration bezeichnet man P1, P2 und ADR als Records. Ein Record besitzt einzelne Felder, auf die man durch Angabe ihres Namens nach einem Punkt zugreift:

```
P1.X:= 30;
P1.Y:= 40;
ADR.ORT      := 'MONOPOLY';
ADR.STRASSE:= 'Schloßallee';
ADR.HAUSNR := 4;
```

Andererseits kann man aber auch den Record als Ganzes ansprechen:

```
P1:= P2
```

Durch diese Zuweisung werden alle

```
PROGRAM UMSAETZE (INPUT, OUTPUT);

  TYPE MONAT = 1..12;
       JAHR  = 1980..1985;
       UMSATZTABELLE: ARRAY [JAHR, MONAT] OF REAL;

  VAR  FARBEN, LACKE: UMSATZTABELLE;

  FUNCTION JAHRESSUMME( J: JAHR; VAR T: UMSATZTABELLE): REAL;
  (* BERECHNE DIE SUMME DER MONATSUMSÄTZE IM JAHR J IN DER
  TABELLE T: *)

   VAR M: MONAT;
       S: REAL;

 BEGIN
   S:= 0;
   FOR M:= 1 TO 12 DO S:= S + T[J,M];
   JAHRESSUMME:= S
 END; (* JAHRESSUMME *)

 FUNCTION MONATSSUMME( M: MONAT; VAR T: UMSATZTABELLE): REAL;
 (* BERECHNE DIE SUMME ALLER UMSÄTZE IM MONAT M FÜR 1980
 BIS 1985 *)

   VAR J: JAHR;
       S: REAL;

  BEGIN
    S:= 0;
    FOR J:= 1980 TO 1985 DO S:= S + T[J,M];
    MONATSSUMME:= S
  END; (* MONATSSUMME *)

  PROCEDURE GESAMTSUMME (VAR T: UMSATZTABELLE): REAL;
   (* BERECHNE DIE SUMME ALLER UMSÄTZE *)

    VAR J: JAHR;
        S: REAL;

  BEGIN
    S:= 0;
    FOR J:= 1980 TO 1985 DO S:= S + JAHRESSUMME(J,T);
    GESAMTSUMME:= S
  END; (* GESAMTSUMME *)

BEGIN

  (* HIER WERDEN DIE UMSÄTZE FÜR FARBEN UND LACKE VORBELEGT *)

  WRITELN('UMSATZ FARBEN 1980:', JAHRESSUMME(1980, FARBEN));
  WRITELN('UMSÄTZE LACKE JANUAR 1980-1985:'); MONATSSUME(1,
  LACKE); WRITELN('UMSÄTZE LACKE UND FARBEN INSGESAMT:',
          GESAMTSUMME(LACKE) + GESAMTSUMME (FARBEN));
END.
```

Listing 14. Bearbeitung zweidimensionaler Tabellen

Felder im Record übertragen. Wichtig und charakteristisch für das Typkonzept in Pascal ist die Tatsache, daß zusammengesetzte Typen auch mehrstufig aufzubauen sind:

```
VAR POSITIONEN: ARRAY [1...99] OF
                        POSITION;
```

Den umgekehrten Fall, daß ein Record aus Arrays besteht, finden Sie bereits im Beispiel des Typs ADRESSE. Dort sind die Felder NAME und VORNAME vom Typ STRING, den wir ja als

```
TYPE STRING = ARRAY [1..11]
OF CHAR;
```

vereinbart hatten. Sollen viele Operationen mit den Feldern eines Records durchgeführt werden, so ist die ständige Wiederholung des Variablennamens lästig. Für diesen Fall existiert die WITH-Anweisung:

```
WITH Recordvariable DO Anweisung
```

So kann man folgende Zuweisungen

```
ADR.NAME   :='Hugendubel';
ADR.VORNAME:='Kunigunde';
ADR.ORT    :='7 Berge';
ADR.STRASSE:='7 Zwerge';
```

schreiben als:

```
WITH ADR DO
BEGIN
   NAME   := 'Hugendubel';
   VORNAME:= 'Kunigunde';
   ORT    := '7 Berge';
   STRASSE:= '7 Zwerge';
END;
```

Wie alles in Pascal können auch WITH-Anweisungen geschachtelt werden. Nach der geschachtelten Record-Deklaration

```
TYPE DATUM = RECORD
                TAG   :  1..31;
                MONAT :  1..12;
                JAHR  :  INTEGER;
              END;
BANKAUSZUG = RECORD
          DATUM: DATUM;
          KONTOSTAND: REAL
             END;
VAR MEINBANKAUSZUG: BANKAUSZUG;
```

prüft man folgendermaßen das Datum auf dem Bankauszug:

```
WITH MEINBANKAUSZUG DO
BEGIN
 IF KONTOSTAND >0 THEN
  WITH DATUM DO
   WRITELN(TAG,'.',MONAT,'.',
JAHR)
END;
```

Dabei muß sich die innere WITH-Anweisung auf ein Feld in dem Record der äußeren WITH-Anweisung beziehen. Hier ist dies also das Feld MEIN BANKAUSZUG.DATUM. Bemerkenswert sind noch die Sichtbarkeitsregeln für Feldnamen. Man kann ohne Namenskonflikte Variablen mit dem Feldnamen aus der Record-Deklaration definieren. Der Feldname ist durch einen Punkt oder die WITH-Anweisung an den Variablennamen des Records gebunden.

Am besten verdeutlicht wiederum ein Beispiel die Arbeit mit Records. In Listing 15 wird der Typ

```
TYPE BRUCH = RECORD
        ZAEHLER, NENNER: INTEGER;
      END;
```

definiert. Man möchte also auch gebrochene Zahlen »exakt« darstellen. Nun lernt man aber in der Schule, daß

$$\frac{1}{2} \quad \text{und} \quad \frac{2}{4} \quad \text{und} \quad \frac{123456}{245912}$$

dieselbe Zahl darstellen. Deshalb werden bei allen Operationen Zähler und Nenner gekürzt dargestellt. Zum Kürzen muß man den größten gemeinsamen Teiler (ggT) von Zähler und Nenner kennen. Beim Addieren werden die beiden Brüche auf einen Hauptnenner gebracht, der natürlich möglichst klein bleiben soll. Daher ist im Programm auch eine Funktion zur Berechnung des kleinsten gemeinsamen Vielfachen (kgV) enthalten. Nach der Erweiterung der beiden Zähler ergibt sich durch Addieren der Zähler des Summenbruches. Wegen der teilerfremden Ausgangsbrüche und der Wahl des Hauptnenners spart man es sich, den Zähler gegen den Nenner zu kürzen.

Die Multiplikation verläuft wie in der Schule nach der Regel Zähler mal Zähler und Nenner mal Nenner. Jedoch werden zuvor die Brüche kreuzweise gekürzt, um die Produkte möglichst klein zu halten. Vielleicht ist es eine gute Denksportaufgabe, zu überlegen, warum nach der Multiplikation Zähler und Nenner teilerfremd sind.

Die Subtraktion ist einfach auf die Addition zurückzuführen, ebenso wie die Division durch Kehrwertbildung auf die Multiplikation zurückführt. Bei der Berechnung des Kehrwertes ist jedoch auf eine korrekte Behandlung des Vorzeichens zu achten.

Normalerweise eignen sich zur Erläuterung von Records Beispiele aus der kaufmännischen Datenverarbeitung, da man einen Record am einfachsten als ein Formblatt beschreiben kann, das verschiedene Felder enthält, die nur Werte gewisser Typen beinhalten dürfen.

```
TYPE KRAFTFAHRZEUGSCHEIN =
  RECORD
    WAGEN: KENNZEICHEN;
    WOHNORT, STANDORT: ADRESSE;
    LEISTUNG: INTEGER;
  END;
```

Diese Deklaration setzt natürlich voraus, daß zuvor die Typen KENNZEICHEN und ADRESSE (zum Beispiel selbst als Records) definiert wurden.

Es gibt jedoch auch Felder, die nur dann gültig sind, falls in einem anderen Feld ein bestimmter Wert steht.

```
TYPE PERSONALAKTE =
   RECORD
     NAME    :  STRING;
     VORNAME:  STRING;
     CASE STAND: FAMILIENSTAND OF
       LEDIG:  ();
       VERHEIRATET,GETRENNT:
         (HEIRAT: DATUM);
       GESCHIEDEN:
         (SCHEIDUNG: DATUM;
          ALIMENTE : BOOLEAN);
   END;
VAR AKTE1: PERSONALAKTE;
```

Dieses Beispiel zeigt einen varianten (veränderlichen) Record. Er besteht aus einem festen Teil (den Feldern NAME und VORNAME), dem ein veränderbarer Teil folgt. Dieser Teil wird durch das Schlüsselwort CASE eingeleitet. Ihm folgt ein Feldname mit Typangabe. In Abhängigkeit der Werte dieses skalaren Typs besitzen die nachfolgenden Feldlisten in runden Klammern Gültigkeit.

Nach der Zuweisung »AKTE1.STAND: = LEDIG« sind nur die Felder NAME und VORNAME gültig. Ist jedoch »AKTE1.STAND = VERHEIRATET«, so wird zusätzlich das Feld HEIRAT vom Typ DATUM relevant. Ihm darf man jetzt Werte des korrekten Typs zuweisen:

```
AKTE1. HEIRAT.TAG  :=7;
AKTE1. HEIRAT.MONAT:=6;
AKTE1. HEIRAT.JAHR :=1973;
```

Dieselben Felder gelten auch für »STAND = GETRENNT«. Sollte im Laufe der Ehegeschichte eine Scheidung eintreten, so werden sich auch in der Personalakte gravierende Änderungen

```
PROGRAM BRUECHE (INPUT, OUTPUT);
(* RECHNUNG MIT BRUECHEN IN DER DARSTELLUNG ZAEHLER, NENNER *)
TYPE BRUCH = RECORD
                ZAEHLER: INTEGER;
                NENNER : INTEGER;
                (* ZAEHLER UND NENNER IMMER    *)
                (* TEILERFREMD, NENNER POSITIV *)
              END;
VAR A, B, SUMME, DIFFERENZ, PRODUKT, QUOTIENT: BRUCH;

  PROCEDURE KUERZE(VAR A,B: INTEGER);
  (* KUERZE A UND B DURCH IHREN GROESSTEN GEMEINSAMEN TEILER *)
    VAR X: INTEGER;

    FUNCTION GGT (X,Y: INTEGER): INTEGER;
    (* BERECHNE DEN GRÖSSTEN GEMEINSAMEN TEILER VON X UND Y *)
      VAR H: INTEGER;
    BEGIN
      IF Y>X THEN
        BEGIN H:=X; X:=Y; Y:=H END;
      REPEAT
        H:= X MOD Y;
        X:= Y; Y:= H
      UNTIL H = 0;
      GGT:= X;
    END; (* GGT *)

  BEGIN (* KÜRZE *)
    X:= GGT(ABS(A),B);
    IF X<>1 THEN
      BEGIN
        A:= A DIV X;
        B:= B DIV X;
      END;
  END; (* KÜRZE *)

  FUNCTION KGV(X,Y: INTEGER): INTEGER;
  (* BERECHNE DAS KLEINSTE GEMEINSAME VIELFACHE VON X UND Y *)

    VAR U,V: INTEGER;

  BEGIN
    U:= X; V:= Y;
    WHILE X<>Y DO
      IF X>Y THEN
        BEGIN X:= X-Y; U:= U+V END
      ELSE
        BEGIN Y:= Y-X; V:= V+U END;

   KGV:= (U+V) DIV 2
   (* ÜBRIGENS IST GGT(X,Y) = X *)
  END; (* KGV *)

PROCEDURE ADDIERE (A,B: BRUCH; VAR C: BRUCH);
(* ADDIERE DIE BRÜCHE A UND B ZUM BRUCH C *)

   VAR HN: INTEGER; (* HAUPTNENNER *)

BEGIN
  WITH C DO
    BEGIN
      (* HAUPTNENNER BERECHNEN *)
      HN := KGV(A.NENNER, B.NENNER);
      NENNER := HN;
      (* AUF HAUPTNENNER BRINGEN *)
      ZAEHLER := A.ZAEHLER * (HN DIV A.NENNER) + B.ZAEHLER *
                 (HN DIV B.NENNER);
    END; (* WITH *)
END; (* ADDIERE *)

PROCEDURE SUBTRAHIERE(A,B: BRUCH; VAR C: BRUCH);
(* SUBTRAHIERE B VON A. ERGEBNIS IN C *)

BEGIN
  B.ZAEHLER:= - B.ZAEHLER;
  ADDIERE(A,B,C)
END; (* SUBTRAHIERE *)

PROCEDURE MULTIPLIZIERE(A,B: BRUCH; VAR C: BRUCH);
(* ZAEHLER MAL ZAEHLER UND NENNER MAL NENNER. VOR DER *)
(* MULTIPLIKATION WERDEN ZAEHLER UND NENNER GEKUERZT. *)

BEGIN
  (* KUERZE ZUNAECHST KREUZWEISE, DAMIT PRODUKT NICHT *)
  (* ZU GROSS WIRD                                     *)
  KUERZE (A.ZAEHLER, B.NENNER);
  KUERZE (B.ZAEHLER, A.NENNER);

  C.ZAEHLER:= A.ZAEHLER * B.ZAEHLER;
  C.NENNER := A.NENNER  * B.NENNER;

  (* ZÄHLER UND NENNER VON C SIND HIER NOCH TEILERFREMD *)
END; (* MULTIPLIZIERE *)

PROCEDURE KEHRWERT(VAR A: BRUCH);
(* BERECHNE DEN KEHRWERT VON A. BEACHTE DIVISION DURCH 0
UND VORZEICHEN *)

  VAR T: INTEGER;

BEGIN
  T:= A.ZAEHLER;
  IF T=0 THEN WRITELN(' DIVISION DURCH NULL!')
    ELSE
      IF T>0 THEN
        BEGIN
          A.ZAEHLER:= A.NENNER;
          A.NENNER := T
        END
      ELSE
        BEGIN
          A.ZAEHLER:= -A.NENNER;
          A.NENNER := -T;
        END;
END; (* KEHRWERT *)

PROCEDURE LESEN (VAR A: BRUCH);
(* LIEST EINEN BRUCH VON DER TASTATUR. WIRD NACH *)
(* DEM ZAEHLER DAS ZEILENENDE ERREICHT, SO WIRD  *)
(* DER NENNER GLEICH EINS GESETZT                *)

 BEGIN
   WITH A DO
     BEGIN
       READ(A.ZAEHLER);
       IF EOLN THEN A.NENNER:= 1
                ELSE READLN(A.NENNER);
     END;
   KUERZE(A.ZAEHLER,A.NENNER);
 END; (* LESEN *)

 PROCEDURE DRUCKEN (A: BRUCH);

 BEGIN
   WITH A DO
     IF NENNER = 1 THEN WRITELN(ZAEHLER)
     ELSE
       WRITELN(ZAEHLER, '/', NENNER)
 END; (* DRUCKEN *)

BEGIN
 WRITELN('RECHNEN MIT BRUECHEN: (ENDE MIT A = 0)');

 REPEAT
   WRITE('A = '); LESEN(A);
   WRITE('B = '); LESEN(B);

   ADDIERE(A,B,SUMME);
   WRITE('A + B = '); DRUCKEN(SUMME);
   SUBTRAHIERE(A,B,DIFFERENZ);
   WRITE('A - B = '); DRUCKEN(DIFFERENZ);'
   MULTIPLIZIERE(A,B,PRODUKT);
   WRITE('A * B = '); DRUCKEN(PRODUKT);
   KEHRWERT(B); MULTIPLIZIERE(A,B,QUOTIENT);
   WRITE('A / B = '); DRUCKEN(QUOTIENT);
 UNTIL A.ZAEHLER = 0;
END.
```

Listing 15. Bruchrechnung mit Records in Pascal

vollziehen. Mit »AKTE1.STAND := GESCHIEDEN« wird das Feld HEIRAT ungültig und statt dessen das Feld SCHEIDUNG (ebenfalls ein Datum) mit dem booleschen Feld ALIMENTE wirksam.

Offensichtlich kann man mit varianten Records sehr gut die Tatsache wiedergeben, daß gewisse Attribute eines Objektes nur unter gewissen Randbedingungen relevant sind. Dies wird dadurch erreicht, daß zu jedem Zeitpunkt immer nur eine der Feldlisten in Klammern hinter den Konstanten gültig ist. Diese zusätzlichen Informationen nutzt der Pascal-Compiler, um wiederum Speicherplatz zu sparen. Er legt alle Varianten (also alle Feldlisten, die zu verschiedenen Werten des Auswahlfeldes gehören) an dieselbe Speicherposition. Zur Laufzeit bestimmt dann das Auswahlfeld nach CASE (engl. tagfield), wie der Inhalt des Speichers zu interpretieren ist. Alle gemeinsamen Felder aus dem festen Teil sind eindeutig, während die Felder im varianten Teil sich gegenseitig überlappen.

Diese Überschneidung muß beachtet werden, wenn ein Wechsel des Auswahlfeldes anfällt. Eine Anwendung varianter Records zeigt Listing 16. Bekanntermaßen kann man einen Punkt in der Ebene auf zwei verschiedene Methoden darstellen:

In rechtwinkligen X- und Y-Koordinaten oder mit Polarkoordinaten, durch Angabe der Entfernung des Punktes zum Koordinatensprung (L) und des Winkels zur X-Achse (PHI).

Da sich eine Darstellungsform für gewisse Anwendungen jeweils besser eignet, wird in diesem Programm mit varianten Records ein Wechsel zwischen beiden Darstellungen zugelassen:

```
TYPE KOORDINATE =
  RECORD
   CASE TYP:KOORDINATENTYP OF
     RECHTWINKLIG:
        (X,Y: REAL);
     POLAR:
        (L,PHI: REAL);
  END;
```

Es ist zu beachten, daß die Angabe des Typs des Auswahlfeldes (hier KOORDINATENTYP) durch einen Typnamen erfolgen muß. Je nach dem Wert von Typ ist also die Koordinate durch X und Y oder L und PHI bestimmt. Im übrigen Programm können Sie neben dem Nutzen der WITH-Anweisung bei der Arbeit mit Records auch den Vorteil der Case-Anweisung bei varianten Records erkennen. Indem man vor dem Zugriff auf eine Variante eine CASE-Anweisung setzt, die die Konstanten aus der Record-Deklaration wiederholt, ist man vor unerlaubten Zuweisungen wie

```
P.TYP:= POLAR;
P.X  := 39.4;
```

geschützt. Erwähnenswert ist noch die Ende-Bedingung der Repeat-Schleife

```
UNTIL BETRAG(P) <= EPSILON;
```

Das Programm soll beendet werden, wenn die Entfernung des Punktes vom Ursprung Null ist. Da jedoch alle reellen Zahlen nur mit Rundungsfehlern berechnet werden können, verwendet man zwischen reellen Zahlen nie den Test auf Gleichheit (A=B), sondern nur eine Prüfung der Form ABS(A-B) <= EPSILON mit einer Konstanten EPSILON, die eine Schranke für die Rundungsfehler auf dem jeweiligen Rechner angibt. Die Berücksichtigung dieser Regel hilft eine Reihe häufiger Fehler zu vermeiden.

Nun sind wir am Ende unserer Einführung in das Programmieren mit Funktionen, Prozeduren und Datentypen angelangt. Zur Vertiefung der gewonnenen Erkenntnisse sei es dringend angeraten, ein wenig mit den abgedruckten Beispielen herumzuexperimentieren, sie zu erweitern und abzuändern. Denn auch in Pascal gilt der altbekannte Spruch vom Meister, den man nur durch Übung . . .

(Florian Matthes/er)

```
PROGRAM KOORDINATEN (INPUT, OUTPUT);
(* RECHNUNG MIT POLAR- UND RECHTWINKLIGEN KOORDINATEN,        *)
(* DARGESTELLT DURCH VARIANTE RECORDS IN PASCAL.              *)
CONST EPSILON = 1.0E-7;                   (* RUNDUNGSFEHLERGRENZE *)

TYPE  KOORDINATENTYP = (RECHTWINKLIG, POLAR);
      KOORDINATE = RECORD
                     CASE TYP: KOORDINATENTYP OF
                       RECHTWINKLIG: (X,Y  : REAL);
                       POLAR       : (L,PHI: REAL);
                    END;

VAR  P: KOORDINATE;

PROCEDURE HOLEPUNKT (VAR P: KOORDINATE);

BEGIN
  REPEAT
    WRITE('C-ARTESISCH ODER P-OLAR ?');
    READLN(CH)
  UNTIL CH IN ['C','P'];

  WITH P DO
    IF CH = 'C' THEN
      BEGIN
        TYP:= RECHTWINKLIG;
        WRITE('X = '); READLN(X);
        WRITE('Y = '); READLN(Y)
      END
    ELSE
      BEGIN
        TYP.= CARTESISCH;
        WRITE('L   = '); READLN(L);  (* ABSTAND VOM KOORDINATEN
        *) URSPRUNG
       WRITE('PHI = '); READLN(PHI); (* WINKEL ZUR X-ACHSE *)
     END
 END; (* HOLEPUNKT *)

 PROCEDURE UMRECHNUNG (VAR P: KOORDINATE);
 (* UMRECHNUNG DER BEIDEN KOORDINATENTYPEN *)
   VAR R: REAL;

 BEGIN
   WITH P DO
     CASE TYP OF
       RECHTWINKLIG:
          BEGIN
            R  := SQRT(X*X + Y*Y);
            PHI:= ARCTAN(Y/X);
            L  := R;
            TYP:= POLAR;
          END;
       POLAR;
          BEGIN
            R  := SIN(PHI)*L;
            Y  := COS(PHI)*L;
            X  := R;
            TYP:= RECHTWINKLIG
          END;
     END; (* CASE & WITH *)
 END: (* UMRECHNUNG *)

 FUNCTION BETRAG( P: KOORDINATE): REAL;
 BEGIN
   IF P.TYP = RECHTWINKLIG THEN
     BETRAG:= SQRT( SQR(P.X) + SQR(P.Y) )
   ELSE
     BETRAG:= P.L
 END; (* BETRAG *)

 PROCEDURE DRUCKEPUNKT (P: KOORDINATE);
 BEGIN
   WITH P DO
     CASE TYP OF
        RECHTWINKLIG: WRITELN('X = ', X, ', Y = ', Y);
        POLAR       : WRITELN('L = ', L, ', PHI = ',PHI);
     END; (* CASE *)
 END; (* DRUCKEPUNKT *)
 BEGIN
   REPEAT
     HOLEPUNKT(P);
     DRUCKEPUNKT(P);
     UMRECHNUNG(P);
     DRUCKEPUNKT(P);
   UNTIL BETRAG(P)<= EPSILON;
 END.
```

Listing 16. Koordinatenumrechnung mit varianten Records

Spitzen-Software für Schneider-Computer und Commodore 128 PC

BRANDNEU
Jetzt auch für den Schneider Joyce

WordStar 3.0 mit MailMerge Der Bestseller unter den Textverarbeitungsprogrammen für PCs bietet Ihnen bildschirmorientierte Formatierung, deutschen Zeichensatz und DIN-Tastatur sowie integrierte Hilfstexte. Mit MailMerge können Sie Serienbriefe mit persönlicher Anrede an eine beliebige Anzahl von Adressen schreiben und auch die Adreßaufkleber drucken.
WordStar/MailMerge für den Schneider CPC 464*, CPC 664*
Bestell-Nr. MS 101 (3"-Diskette)
Bestell-Nr. MS 102 (5¼"-Diskette im VORTEX-Format)
WordStar/MailMerge für den Schneider CPC 6128
Bestell-Nr. MS 104 (3"-Diskette)
WordStar/MailMerge für den Schneider Joyce PCW 8256
Best.-Nr. MS 105 (3"-Diskette)
Hardware-Anforderungen: Schneider CPC 464*, CPC 664*, CPC 6128 oder Joyce, beliebiger Drucker mit Centronics-Schnittstelle
* Der Standard-Speicherplatz beim CPC 464/664 erlaubt ohne Speichererweiterung Blockverschiebe-Operationen nur bedingt und Simultan-Drucken gar nicht.
WordStar/MailMerge für den Commodore 128 PC
Bestell-Nr. MS 103 (5¼"-Diskette)
Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Markt&Technik
Schneider CPC Software



mit MailMerge für den Schneider CPC 464/664
3" Schneider-Format



Und dazu die weiterführende Literatur:

WordStar für den Schneider CPC
Best.-Nr. MT 779, ISBN 3-89090-180-8
WordStar für den Commodore 128 PC
Best.-Nr. MT 780, ISBN 3-89090-181-6

dBASE II, Version 2.41 dBASE II, das meistverkaufte Programm unter den Datenbanksystemen, eröffnet Ihnen optimale Möglichkeiten der Daten- u. Dateihandhabung. Einfach u. schnell können Datenstrukturen definiert, benutzt und geändert werden. Der Datenzugriff erfolgt sequentiell oder nach frei wählbaren Kriterien, die integrierte Kommandosprache ermöglicht den Aufbau kompletter Anwendungen wie Finanzbuchhaltung, Lagerverwaltung, Betriebsabrechnung usw.
dBASE II für den Schneider CPC 464*, CPC 664*
Bestell-Nr. MS 301 (3"-Diskette)
Bestell-Nr. MS 302 (5¼"-Diskette im VORTEX-Format)
dBASE II für den Schneider CPC 6128
Bestell-Nr. MS 304 (3"-Diskette)
dBASE II für den Schneider Joyce PCW 8256
Best.-Nr. MS 305 (3"-Diskette)
Hardware-Anforderungen: Schneider CPC 464*, CPC 664*, CPC 6128 oder Joyce, beliebiger Drucker mit Centronics-Schnittstelle
* dBASE II für den Schneider CPC 464/664 ist lauffähig mit der VORTEX-Speichererweiterung auf 128 KByte. Diese erhalten Sie direkt bei der Firma VORTEX oder bei Ihrem Computerhändler.
dBASE II für den Commodore 128 PC
Bestell-Nr. MS 303 (5¼"-Diskette)
Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Markt&Technik
Schneider CPC Software
dBASE™ II
ASHTON-TATE
für den Schneider CPC 6128
3" Schneider-Format



dBASE II für den Schneider CPC
Best.-Nr. MT 837, ISBN 3-89090-188-3
dBASE II für den Commodore 128 PC
Best.-Nr. MT 838, ISBN 3-89090-189-1

MULTIPLAN, Version 1.06 Wenn Sie die zeitraubende manuelle Verwaltung tabellarischer Aufstellungen mit Bleistift, Radiergummi und Rechenmaschine satt haben, dann ist MULTIPLAN, das System zur Bearbeitung »elektronischer Datenblätter«, genau das richtige für Sie! Das benutzerfreundliche und leistungsfähige Tabellenkalkulationsprogramm kann bei allen Analyse- und Planungsberechnungen eingesetzt werden wie z.B. Budgetplanungen, Produktkalkulationen, Personalkosten usw. Spezielle Formatierungs-, Aufbereitungs- und Druckanweisungen ermöglichen außerdem optimal aufbereitete Präsentationsunterlagen!
MULTIPLAN für den Schneider CPC 464*, CPC 664*
Bestell-Nr. MS 201 (3"-Diskette)
Bestell-Nr. MS 202 (5¼"-Diskette im VORTEX-Format)
MULTIPLAN für den Schneider CPC 6128
Bestell-Nr. MS 204 (3"-Diskette)
MULTIPLAN für den Schneider Joyce PCW 8256
Best.-Nr. MS 205 ("-Diskette)
Hardware-Anforderungen: Schneider CPC 464*, CPC 664*, CPC 6128 oder Joyce, beliebiger Drucker mit Centronics-Schnittstelle
* MULTIPLAN für den Schneider CPC 464/664 ist lauffähig mit der VORTEX-Speichererweiterung auf 128 KByte.
MULTIPLAN für den Commodore 128 PC
Bestell-Nr. MS 203 (5¼"-Diskette)
Hardware-Anforderungen: Commodore 128 PC, Diskettenlaufwerk, 80-Zeichen-Monitor, beliebiger Commodore-Drucker oder ein Drucker mit Centronics-Schnittstelle

Markt&Technik
128er-Software
MICROSOFT MULTIPLAN
für den Commodore 128 PC
5¼"-Diskette im Floppy 1541-Format

MULTIPLAN für den Commodore 128 PC

MULTIPLAN für den Schneider CPC
Best.-Nr. MT 835, ISBN 3-89090-186-7
MULTIPLAN für den Commodore 128 PC
Best.-Nr. MT 836, ISBN 3-89090-187-5

Jedes Buch kostet DM 49,-
(sFr. 45,10/öS 382,20).
Erhältlich bei Ihrem Buchhändler.

Sie erhalten jedes **WordStar-, dBASE II- und MULTIPLAN-Programm** für Ihren Schneider-Computer oder Commodore 128 PC fertig angepaßt (Bildschirmsteuerung). **Jeweils Originalprodukte!** Jedes Programmpaket enthält außerdem ein ausführliches Handbuch mit kompakter Befehlsübersicht. Die VORTEX-Speichererweiterung für den Schneider CPC 464 erhalten Sie direkt bei der Firma VORTEX oder bei Ihrem Computerhändler.

Diese Markt & Technik-Softwareprodukte erhalten Sie in den Computer-Abteilungen der Kaufhäuser, bei Ihrem Computerhändler oder im Buchhandel.

Wenn Sie direkt beim Verlag bestellen wollen: gegen Vorauskasse durch Verrechnungsscheck oder mit der eingehefteten Zahlkarte.

Bestellungen im Ausland bitte an untenstehende Adressen.
Schweiz: Markt & Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, ☎ 042/41 56 56
Österreich: Überreuter Media, Handels- und Verlagsges. mbH, Alser-Str. 24, A-1091 Wien, ☎ 0222/48 15 38-0

Für Auskünfte steht Ihnen Herr Teller, Telefon 089/46 13-205, gerne zur Verfügung.

Jedes Programm kostet DM 199,-* (sFr. 178,-/öS 1890,-*) * Inkl. MwSt. Unverbindliche Preisempfehlung



Unternehmensbereich Buchverlag
Hans-Pinsel-Straße 2, 8013 Haar bei München

Dateiverwaltung mit Pascal

Hat man große Datenmengen zu verarbeiten, so reicht der Arbeitsspeicher des Computers im allgemeinen nicht aus. Außerdem gehen alle Daten im Hauptspeicher beim Ausschalten des Computers verloren. Um langfristig große Datenbestände zu konservieren, benötigt man externe Speichermedien (Floppy-Disk, Festplatten, aber auch Band- und Kassettenlaufwerke).

Ein großes Problem ist nun die Tatsache, daß es fast so viele Betriebssysteme wie Computertypen gibt. Da jedoch Pascal sowohl auf Mikrocomputern als auch Großrechenanlagen implementiert ist, muß ein möglichst systemunabhängiges Konzept zur Darstellung externer Speicher in der Sprache gefunden werden. Diese Lösung besteht in der Formalisierung des Prinzips der sequentiellen Dateien, die man als den kleinsten gemeinsamen Nenner aller Dateiverwaltungssysteme bezeichnen kann.

Mit der Deklaration

```
TYPE ZAHLENDATEI = FILE OF
INTEGER;
VAR D: ZAHLENDATEI;
```

wird ein File D vereinbart, dessen Komponenten aus ganzen Zahlen bestehen. Allgemein kann nach den Schlüsselworten FILE und OF jeder beliebige (auch zusammengesetzte) Typ folgen. Wie in einem Array besitzen alle Komponenten denselben Typ. Jedoch kann ein File (praktisch) beliebig viele Komponenten umfassen, die nur in fester Reihenfolge adressiert werden können. Zu jedem Zeitpunkt ist immer nur eine Komponente erreichbar. Diese aktuelle Komponente wird im obigen Beispiel mit D↑ bezeichnet und ist vom Typ INTEGER.

Um nun weitere Komponenten zu erreichen, kann man die aktuelle Komponente (Puffervariable) D↑ wie ein Fenster über das gesamte File verschieben (siehe Bild 1). Ein konkretes Beispiel für das Schreiben und Lesen eines Files zeigt Listing 1. Es demonstriert die beiden Methoden des Zugriffs auf Files in Pascal.

Dieser Beitrag beschäftigt sich mit dem Datei-Begriff von Pascal und gibt Hinweise und Beispiele für die effiziente Verwaltung großer Datenmengen.

1. Sequentielles Schreiben

Um ein File zu erzeugen, muß man die Standardprozedur REWRITE mit der Filevariablen (im Beispiel also D↑) aufrufen. Dadurch werden alle Komponenten gelöscht, die eventuell zuvor im File existierten. Das Schreibfenster (D↑ im Beispiel) steht über der ersten Komponente und besitzt einen undefinierten Wert (siehe Bild 2a). Eine Zuweisung wie

`D↑ := 99;`

füllt den Puffer mit dem angegebenen Wert (Bild 2b). Natürlich erfolgen auch bei Files die üblichen Typüberprüfungen, so daß der Compiler die Zuweisung »D↑ :='K'« als fehlerhaft erkennt. Mit PUT(D) wird der Wert des Puffers in das File geschrieben und der Schreibzeiger eine Position weiter gesetzt. Jede Wiederholung der Schritte Wertzuweisung an den Puffer und Puffer schreiben erweiter das File um jeweils eine Komponente (Bild 2c):

`D↑:= 43; PUT(D);`

Es ist beim sequentiellen Schreiben nicht möglich, eine bereits geschriebene Komponente wieder zu korrigieren, da dazu der Schreibzeiger rückwärts bewegt werden müßte.

2. Sequentielles Lesen

Durch den Aufruf der Standardprozedur RESET mit der Filevariablen als Parameter wird die Puffervariable auf den Wert der ersten Komponente im File gesetzt (Bild 2d). Die Puffervariable kann man wie eine »normale« Variable in Ausdrücken verwenden und so das File Stück für Stück bearbeiten. Im Beispiel ergibt die Befehlsfolge

`RESET(D); WRITE(D↑, D↑+5)`

die Ausgabe »99 104«. Aufruf GET(D) setzt die Puffervariable wiederum eine Position weiter, so daß D↑ den Wert 43 enthält (Bild 2 e). Nach einem erneuten Aufruf von GET(D) steht das Lesefenster hinter der letzten belegten Komponente (Bild 2f) und die Puffervariable ist undefiniert. Um festzustellen, ob das Lesefenster über einem definierten Wert steht oder sich hinter dem Fileende befindet, dient die Standardfunktion

`EOF(Filevariable)`

So lieferte EOF(D) den booleschen Wert TRUE, falls D↑ sich in der letzten Komponente (wie in Bild 2f) befindet. Insbesondere ist EOF=TRUE, falls man ein leeres File mit RESET zum Lesen eröffnet. Da im Beispielprogramm in Listing 1 die Länge des Files unbekannt ist, wird eine Schleife der Form

`WHILE NOT EOF(D) DO`

verwendet. Übrigens ist per Definition EOF(D)=TRUE, falls auf das File D (nach RESET(D)) geschrieben wird.

Man kann ein File in beliebiger Reihenfolge mit RESET und REWRITE zum sequentiellen Lesen und Schreiben

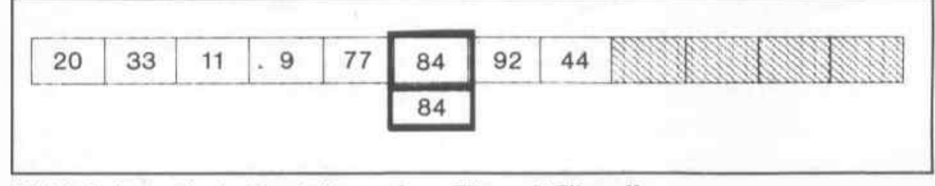


Bild 1. Schematische Darstellung eines Files mit Filepuffer

```
PROGRAM FILES (INPUT, OUTPUT);
  VAR ZAHLENDATEI: FILE OF INTEGER;
      X          : INTEGER;

BEGIN
  (* ZAHLEN VON DER TASTATUR AUF FILE SCHRIEIBEN: *)
  REWRITE(ZAHLENDATEI);          (* SEQUENTIELL SCHREIBEN *)
  REPEAT
    READLN(X);
    ZAHLENDATEI^:= X;            (* PUFFER MIT ZAHL FÜLLEN *)
    PUT(ZAHLENDATEI);            (* PUFFER SCHREIBEN       *)
  UNTIL X=0;

  (* ZAHLEN VOM FILE LESEN UND SCHREIBEN: *)
  RESET(ZAHLENDATEI);            (* SEQUENTIELL LESEN *)
  WHILE NOT EOF(ZAHLENDATEI) DO (* SOLANGE NICHT HINTER DEM DATEIENDE *
    BEGIN
      X:= ZAHLENDATEI^;           (* PUFFER AUSLESEN *)
      WRITELN(X);
      GET(ZAHLENDATEI);           (* PUFFER AUF NÄCHSTE ZAHL *)
    END;
END.
```

Listing 1. Grundlegende Fileoperationen

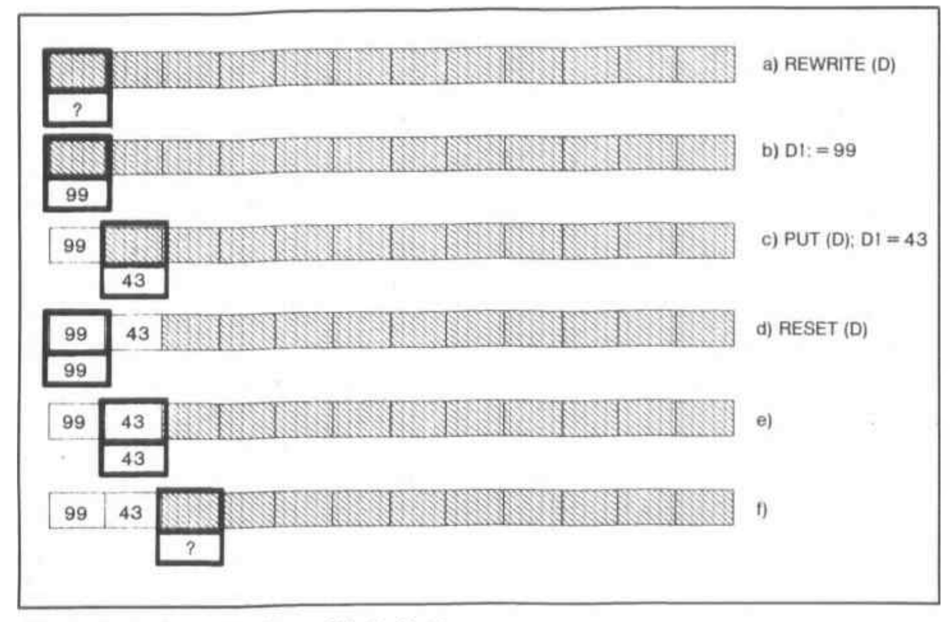


Bild 2. Operationen in einem File (a bis f)

```
PROGRAM MERGE(INPUT, OUTPUT);
(* 2 PHASEN 3-BAND NATÜRLICHES MISCHSORTIERTEN *)

TYPE ITEM = RECORD
              KEY: INTEGER;   (* ODER CHAR, REAL, ARRA[..] OF CHAR *)
              (* HIER KÖNNEN WEITERE FELDER STEHEN *)
            END;

     TAPE = FILE OF ITEM;

VAR  C  : TAPE;          (* DIESES FILE WIRD SORTIERT *)
     BUF: ITEM;          (* HILFSVARIABLE ZUR EINGABE *)


  PROCEDURE LIST (VAR F: TAPE);
  (* ZEIGE DEN INHALT VON F AN *)
    VAR X: ITEM;
  BEGIN
    RESET(F);
    WHILE NOT EOF(F) DO
      BEGIN
        X.KEY:= F^.KEY;
        GET(F);
        WRITE(X.KEY:4)
      END;
    WRITELN
  END; (* LIST *)

PRCEDURE NATURALMERGE;
(* SORITERE DAS FILE C (GLOBAL) IN MEHREREN DURCHLÄUFEN.      *)
(* DIE PROZEDUR BENUTZT ZWEI HILFSFILES A UND B.              *)
  VAR L  : INTEGER; (* ANZAHL DER LÄUFE AUF C                 *)
      EOR: BOOLEAN  (* END OF RUN, ENDE DES LAUFES            *)
      A,B: TAPE;    (* HILFSFILES                             *)

  PROCEDURE COPY(VAR X,Y: TAPE);
  (* KOPIERE RECORD VON X NACH Y, AKTUALISIERE EOR            *)
    VAR BUF: ITEM;
  BEGIN
    BUF.KEY:= X^.KEY; GET(X);
    Y^.KEY:= BUF.KEY; PUT(Y);
    IF EOF(X) THEN EOR:=TRUE
              ELSE EOR:=BUF.KEY>X^.KEY
  END; (* COPY *)

  PROCEDURE COPYRUN (VAR X,Y: TAPE);
  (* KOPIERE KOMPLETTEN LAUF VON X NACH Y *)
  BEGIN
    REPEAT COPY(X,Y) UNTIL EOR
  END; (* COPYRUN *)

  PROCEDURE DISTRIBUTE;
  (* KOPERE LÄUFE VON C ABWECHSELND AUF A UND B *)
  BEGIN
    REPEAT
```

Listing 2. Natürliches Mischsortieren mit Files

eröffnen, jedoch ist zu beachten, daß jeder Aufruf mit REWRITE alle im File existierenden Komponenten löscht. Möchte man ein bestehendes File am Ende erweitern, so muß man zunächst das gesamte File mit GET und PUT auf ein anderes File kopieren, an das man dann mit PUT die neuen Komponenten anfügt. Sollten Sie beim Test des Programmes in Listing 1 nicht das gewünschte Ergebnis erhalten, so liegt das wahrscheinlich daran, daß der Compiler für Ihren Computer zusätzliche Anweisungen zur Arbeit mit Files benötigt.

In Turbo-Pascal zum Beispiel verlangt eine Filevariable vor dem ersten RESET oder REWRITE mit der Prozedur ASSIGN einen

```
ASSIGN(D, 'DATEN.SEQ')
```

Außerdem sollten in Turbo-Pascal unter dem Betriebssystem PC-DOS alle Files (auch solche, von denen nur gelesen wurde) am Ende der Verarbeitung mit CLOSE geschlossen werden:

```
CLOSE(D)
```

Werfen Sie bei Schwierigkeiten also zunächst einen Blick in die Handbücher. Neben diesem zusätzlichen Aufwand bei der Behandlung sequentieller Dateien bieten viele Compiler die Möglichkeit, die Schreib- und Leseposition in einem File gezielt zu setzen. Mit

```
SEEK(D,400)
```

zeigt die Puffervariable auf die 400. Komponente im File. Arbeitet man mit einer Festplatte, so läßt sich ein File wie ein Array variabler Größe mit hundertfach langsamerer Zugriffszeit verwenden.

Zurück zum Pascal-Standard. Eine grundlegende Operation mit Files ist das Sortieren. Für das Sortieren von Arrays gibt es eine Vielzahl von Algorithmen, deren Verständlichkeit oft umgekehrt proportional zur Sortiergeschwindigkeit ist. Falls Sie Pascal und nicht das Sortieren auf externen Speichermedien lernen wollen, ist in Listing 2 ein Programm zum sogenannten »3-Band-, 2-Phasen-Mischsortieren« abgedruckt.

Natürlich kann man ein File dadurch sortieren, daß man es komplett in den Speicher lädt, dort sortiert und es anschließend geordnet zurückschreibt. Reicht jedoch (was häufig der Fall ist) der Arbeitsspeicher nicht aus, so hat man von jedem File in jedem Schritt nur die Komponente der Puffervariablen zur Verfügung. Um ein File umzusortieren, benötigt man also mindestens drei Files. Von einem File wird sequentiell gelesen, während die gelesenen Werte in möglichst geschickter Reihenfolge auf die beiden übrigen Files verteilt werden (Prozedur DISTRIBUTE). Anschließend werden die beiden erzeugten Files zum Lesen eröffnet und auf das

ursprüngliche Lesefile zurückgeschrieben (Prozedur MERGE), wobei jeweils zwei sortierte Teilsequenzen zu einer sortierten Sequenz auf dem Ausgangsfile verbunden werden.

Durch eine Wiederholung der zwei Phasen Verteilen und Mischen (dies ist der Fachausdruck für das Zusammenfassen zweier teilweise geordneter Files zu einem dritten, ebenfalls geordneten File) erhält man schließlich ein sortiertes File. Ein Beispiel soll das Verhalten des Programmes verdeutlichen:

```
C = (1 4 3 2 7 5 6 8)  verteilt ergibt
  A = (1 4 2 7)
  B = (3 5 6 8)     gemischt ergibt
C = (1 3 4 5 6 8 2 7)  verteilt ergibt
  A = (1 3 4 5 6 8)
  B = (2 7)         gemischt ergibt
C = (1 2 3 4 5 6 7 8)
```

Noch ein paar Worte zur Theorie: Eine geordnete Teilfolge innerhalb eines Files wird als Lauf (englisch run) bezeichnet. So besteht die Folge

1,4/3/2,7/5,6,8

aus Läufen. Demzufolge werden mit DISTRIBUTE Läufe von C abwechselnd nach A und B verteilt, und anschließend mit MERGE Läufe auf A und B zu Läufen auf C zusammengefaßt. Die Variable L zählt die Anzahl der Läufe auf C. Ist L=1, so muß also C aufsteigend geordnet sein.

Das Programm selbst zeigt praktisch alle Operationen, die mit Files in Pascal möglich sind. Zunächst wird ein Recordtyp ITEM vereinbart, der ein Schlüsselfeld KEY vom Typ INTEGER besitzt. Natürlich könnte an dieser Stelle auch ein anderer skalarer Typ oder auch der Typ ARRAY[1..N] OF CHAR stehen. Dann interpretiert der Compiler alle Vergleichsoperationen mit »=« und »>« korrekt.

Anschließend wird der eigentliche Filetyp TAPE deklariert. Jede Komponente der Files besteht also aus einem Record, so daß mit jedem Aufruf von PUT und GET ein kompletter Record zwischen Puffervariable und File übertragen wird.

C bezeichnet das zu sortierende File, das am Programmanfang in einer REPEAT-Schleife mit einer Folge ganzer Zahlen gefüllt wird. Die Prozedur LIST zeigt, daß man auch Files als Parameter übergeben kann. Dabei muß man jedoch Variablenmeter verwenden. In LIST wird zunächst das File F zum Lesen eröffnet und anschließend in der üblichen Weise mit einer WHILE-Schleife ausgelesen. Hinter dem Namen NATURALMERGE verbirgt sich die eigentliche Prozedur zum Sortieren. Den Beginn und das Ende des Anweisungsteils der Prozedur kennzeichnen Kommentare. Sicher ist Ihnen schon aufgefallen, daß man ein größeres File in Pascal am besten von hinten liest, da

```
      COPYRUN(C,A);
      IF NOT EOF(C) THEN COPYRUN(C,B);
    UNTIL EOF(C);
  END; (* DISTRIBUTE *)

  PROCEDURE MERGE;
  (* MISCHE FILE A UND B ZU FILE C. L (GLOBAL) ZÄHLT DIE LÄUFE *)

    PROCEDURE MERGERUN;
    (* MISCHE LÄUFE VON A UND B ZU LÄUFEN AUF C *)
    BEGIN
      REPEAT
        IF A^.KEY<B^.KEY THEN      (* KLEINEREN SCHLÜSSEL KOPIEREN *)
          BEGIN
            COPY(A,C);
            IF EOR THEN COPYRUN(B,C)
          END
        ELSE
          BEGIN
            COPY(B,C);
            IF EOR THEN COPYRUN(A,C)
          END
      UNTIL EOR;
    END; (* MERGERUN *)

    BEGIN (* MERGE *)
      L:= 0;
      (* ZUNÄCHST FILES MISCHEN, BIS EINES ZU ENDE: *)
      REPEAT
        MERGERUN; L:=L+1
      UNTIL EOF(A) OR EOF(B);

      (* JETZT DEN REST VOM JEWEILIGEN FILE KOPIEREN *)
      WHILE NOT EOF(A) DO
        BEGIN
          COPYRUN(A,C);
          L:=L+1
        END;
      WHILE NOT EOF(B) DO
        BEGIN
          COPYRUN(B,C);
          L:=L+1
        END;
    END; (* MERGE *)

  BEGIN (* NATURALMERGE *)
    (* ABWECHSELND VERTEILEN UND MISCHEN,  BIS  NUR  EIN LAUF AUF C IST *)
    REPEAT
      REWRITE(A); REWRITE(B); RESET(C);  (* VON C NACH A UND B *)
      DISTRIBUTE;
      RESET(A); RESET(B); REWRITE(C)
      MERGE;
      LIST(C); (* NUR ZUR VERDEUTLICHUNG *)
    UNTIL L=1;
  END; (* NATURALMERGE *)

BEGIN (* HAUPTPROGRAMM *)
  WRITELN('SORTIEREN EINES SEQUENTIELLEN FILES:');
  WRITELN('EINGABEZAHLEN: (MIT 0 ABSCHLIESSEN)');

  REWRITE(C);
  READ(BUF.KEY);
  REPEAT
    C^.KEY:= BUF.KEY; PUT(C);       (* AUF C SCHREIBEN *)
    READ(BUF.KEY)
  UNTIL BUF.KEY=0;

  LIST(C);           (* ZUR KONTROLLE EINGABEZAHLEN ANZEIGEN *)
  NATURALMERGE;      (* SORTIERE C *)
END.
```

Listing 2. Natürliches Mischsortieren mit Files (Schluß)

durch die Schachtelung der Prozeduren die Anweisungsteile des Hauptprogrammes und der wichtigen Prozeduren eben nach »hinten« wandern.

NATURALMERGE selbst besteht, wie bereits besprochen, im wechselseitigen Aufruf der beiden Phasen DISTRIBUTE und MERGE, bis C nur noch einen Lauf umfaßt. In der Hierarchie der Prozeduren steht die Prozedur COPY am weitesten unten. Sie kopiert eine Komponente vom File X zum File Y. Um das Ende eines Laufes zu erkennen, findet hier die boolesche Variable EOR (end of run, Ende des Laufs) Anwendung. So ist nach der Zuweisung

`EOR:= BUF.KEY >X↑.KEY`

die Variable EOR = TRUE, falls das Feld KEY im Record BUF größer als das entsprechende Feld in der Puffervariablen des Files X ist. Da nach jedem Durchlauf die bisher erreichte Sortierung auf dem File C angezeigt wird, sehen Sie mit den folgenden Testdaten die Funktionsweise des Programmes am deutlichsten:

8,7,6,5,4,3,2,1,0

Als nächstes wenden wir uns einem speziellen Typ von Files zu. Während die Files, die mit PUT und GET erzeugt werden, die Daten Byte für Byte wie im Speicher darstellen, ist es oft sinnvoll,

Ein- und Ausgaben zu verwenden, die ein Mensch verstehen kann. Konkret gesprochen stellt man sich unter der Zahl 12345 die Zeichenfolge »12345« und nicht etwa zwei Byte ($30 und $39 hexadezimal) vor.

Da Files, die aus Zeichenfolgen bestehen, eine große Rolle spielen, existiert in Pascal ein vordefinierter Typenbezeichner

```
TYPE TEXT = FILE OF CHAR;
```

Mit der Deklaration

```
VAR F: TEXT;
```

können neben den Operationen RESET und REWRITE auf das Textfile F auch formatiert Werte ein- und ausgegeben werden. Das Prinzip besteht darin, den Prozeduren READ, READLN, WRITE und WRITELN als ersten Parameter eine Filevariable zu übergeben, von der die Eingabe kommt oder auf welche die Ausgabe erfolgen soll:

(1) WRITE(F,AUSDRUCK : n)

Mit diesem Prozeduraufruf wird der Ausdruck vom Typ INTEGER, REAL, CHAR oder BOOLEAN auf das File F in ein Feld mit einer Mindestlänge von n Zeichen geschrieben. Ist die Länge des auszugebenden Wertes größer als n, so wird n ignoriert:

```
WRITE(123456 : 10); WRITE('*': 3)
```

erzeugt also folgende Ausgabe:

```
____123456____*
```

Ist der Ausdruck vom Typ REAL, so sind nach dem Doppelpunkt zwei Formatierungsangaben möglich:

```
WRITE(F,reeller Ausdruck: n : m)
```

druckt die reale Zahl nicht in Gleitkommadarstellung (zum Beispiel 1.00000E+2), sondern in Festkommadarstellung. Dabei gibt n wieder die Mindestgröße des Feldes an, in das die Zahl ausgegeben wird, wohingegen m die Anzahl der angezeigten Nachkommastellen wiedergibt:

```
WRITELN(1.2345 : 10: 5)
```

ergibt die Ausgabe

```
___1.23450
```

(2) WRITELN(F)

Wichtig ist die Tatsache, daß sich die Zeichen auf einem Textfile in Zeilen gliedern. Mit dem Prozeduraufruf WRITELN(F) wird auf dem Textfile F ein Zeilenende erzeugt. Technisch bewirkt dies die Ausgabe von ein oder zwei Steuerzeichen (CR, eventuell LF, carriage return und line feed, also Wagenrücklauf und Zeilenvorschub) auf das File F.

Eine Folge von Ausgaben, wie

```
WRITE(F,A); WRITE(F,B); WRITE(F,
C);
```

kann immer zusammengefaßt werden zu

```
WRITE(F,A,B,C)
```

Außerdem entspricht der Aufruf

```
WRITELN(F,A,B,...)
```

der Befehlsfolge

```
WRITE(F,A,B,...);WRITELN(F)
```

Somit erzeugt also die Ausgabe mit WRITELN immer einen Zeilenwechsel am Ende der Ausgabe. Natürlich muß vor allen Schreiboperationen das File F mit REWRITE(F) zum Schreiben eröffnet worden sein. Dabei werden wieder alle Zeichen, die eventuell zuvor in F existierten, gelöscht. Beabsichtigt man eine Ausgabe auf dem Standard-Ausgabegerät (dem Bildschirm), so bietet sich als Filevariable die vordeklarierte Variable OUTPUT an. Diese kann man sich mit der Deklaration

```
VAR OUTPUT: TEXT;
```

vereinbart denken. In Wirklichkeit wird man jedoch in diesem Fall den Fileparameter bei WRITE nicht angeben, und erhält somit die vereinfachte Syntax, die Sie bereits am Anfang des Artikels kennengelernt hatten:

```
WRITE(OUTPUT, 'Hallo Bildschirm')
```

läßt sich also ersetzen durch

```
WRITE('Hallo Bildschirm').
```

(3) READ(F, Variable)

Vom Textfile F werden Zeichenfolgen eingelesen und als Werte des Typs der Variablen interpretiert. Am besten kann man diese Umwandlung von Zeichenfolgen auf dem File in Werte, zum Beispiel des Typs INTEGER, an Beispielen verdeutlichen:

```
VAR I: INTEGER;
    R: REAL;
    C: CHAR;
```

```
PROGRAM TEXTFILES (INPUT,OUTPUT);

  PROCEDURE FORMAT(VAR EINGABE, AUSGABE: TEXT; RECHTS: INTEGER);
  (* FORMATIERE VON EINGABE NACH AUSGABE AUF RECHTEN RAND IN DER SPALTE  *)
  (* RECHTS. LEERE ZEILEN, ODER ZEILEN MIT NUR EINEM WORT WERDEN NICHT   *)
  (* VERÄNDERT.                                                          *)

  CONST SPACE     = ' ';
  TYPE  LINEINDEX = 1..136;
  VAR   LINE      : ARRAY [LINEINDEX] OF CHAR;
        BIS,I: LINEINDEX;
        ZEILE: INTEGER; (* LFD. ZEILENNUMMER                             *)
        S    : INTEGER; (* ANZAHL DER EINZUFÜGENDEN LEERZEICHEN IN ZEILE *)
        P,Q  : INTEGER; (* ANZAHL DER LEERZEICHEN NACH JEDEM WORT        *)
        T    : INTEGER; (* INDEX DES WORTES, NACH DEM ZUM 1. MAL Q LEER- *)
                        (* ZEICHEN EINGEFÜGT WERDEN SOLLEN               *)
        K,N  : INTEGER; (* ANZAHL DER WORTE IN DER ZEILE                 *)

    PROCEDURE READLINE(VAR LASTCOLUMN: INTEGER; VAR WORDS: INTEGER);
    (* KOMPLETTE ZEILE EINLESEN UND IN LINE (GLOBAL) SPEICHERN   *)
    (* IN LASTCOLUMN STEHT DAS LETZTE ZEICHEN UNGLEICH SPACE IM  *)
    (* TEXT. HINTER DEM TEXT EIN SPACE. WORD ZÄHLT DIE DURCH     *)
    (* LEERZEICHEN GETRENNTEN WORTE.                             *)

     VAR  WASSPACE, ISSPACE: BOOLEAN;
          I                : LINEINDEX;
          CH               : CHAR;
    BEGIN
      I:=1; WASSPACE:= TRUE; WORDS:= 0; LASTCOLUMN:=0;
      WHILE NOT EOLN(EINGABE) DO
        BEGIN
          READ(EINGABE,CH); LINE[I]:= CH;
          ISSPACE:= CH=SPACE;
          IF NOT ISSPACE THEN
            BEGIN
              LASTCOLUMN:= I;
              IF WASSPACE THEN WORDS:= WORDS +1;   (* WORTANFANG *)
            END;
          WASSPACE:= ISSPACE; I:= I+1
        END;
      LINE[LASTCOLUMN+1]:= SPACE; READLN(EINGABE);
    END; (* READLINE *)

    PROCEDURE COPYWORD;
    (* AB DER MOMENTANEN POSITION IN LINE WORT MIT VORLAUFENDEN SPACES *)
    (* BIS ZUM NÄCHSTEN SPACE AUSGEBEN                                 *)
    BEGIN
      WHILE LINE[I]=SPACE DO
        BEGIN WRITE(AUSGABE,SPACE); I:= I+1 END;
      REPEAT
        WRITE(AUSGABE,LINE[I]); I:=I+1
      UNTIL LINE[I]=SPACE;
    END; (* COPYWORD *)

    PROCEDURE INSERTSPACES(N: INTEGER);
    (* N LEERZEICHEN AUSGEBEN *)
    BEGIN
      IF N>0 THEN WRITE (AUSGABE, SPACE : N)
    END; (* INSERTSPACES *)

    BEGIN (* FORMAT *)
      RESET(EINGABE); REWRITE(AUSGABE);
      ZEILE:= 0;

      WHILE NOT EOF(EINGABE) DO
        BEGIN
          READLINE(BIS,N); ZEILE:= ZEILE + 1;
          S:= RECHTS-BIS;    (* ANZAHL DER FEHLENDEN LEERZEICHEN *)
```

Listing 3. Kopieren und Formatieren eines Textfiles

Befindet sich auf dem File F folgende Zeichenfolge:

`1234        34.55X`

so bewirkt der Prozeduraufruf READ (F,I,R,C) folgende Zuweisungen:

`I:=1234; R:=34.55; C:='X'`

Ist der Parameter bei READ eine ganze oder reelle Zahl, so werden zunächst Leerzeichen und Zeilenwechsel ignoriert. Anschließend wird eine Ziffernfolge eingelesen und der entsprechende Wert der Variablen zugewiesen. Eine nachfolgende READ-Operation verarbeitet das Zeichen, das direkt hinter der gelesenen Zahl beginnt. Daher besitzt die Variable C im obigen Beispiel den Wert »X«.

(4) READLN(F)

Vom File F werden solange Zeichen eingelesen, bis ein Zeilenende erkannt wurde. Der nächste Aufruf der Prozedur READ mit dem File F liest das erste Zeichen der folgenden Zeile. Es gibt ebenfalls ein Standardfile zur Eingabe, das folgendermaßen vordeklariert ist:

`VAR INPUT: TEXT;`

Wiederum kann man Eingaben von der Tastatur als Standard-Eingabe auch ohne Angabe einer Filevariablen erreichen. READ(INPUT,I,R,C) entspricht READ(I,R,C) und liest drei Werte von der Tastatur.

(5) EOLN(F)

Die Standardfunktion EOLN, angewandt auf ein Textfile F, liefert einen booleschen Wert. Er ist genau dann TRUE, falls bei der letzten Eingabe vom File F das Zeilenende erreicht wurde. Bitte beachten Sie, daß Sie beim Einlesen nie die oben genannten Steuerzeichen (Zeilenwechsel oder Wagenrücklauf) erhalten, da diese vom Pascal-Laufzeitsystem automatisch in Leerzeichen (blanks) umgewandelt werden.

Nach diesen zugegebenermaßen recht detaillierten Ausführungen über Textfiles in Pascal sollen Sie zur Belohnung auch ein wirklich sinnvolles Programm kennenlernen. Es kopiert einen Text von einem Textfile zu einem anderen und formatiert dabei Zeilen rechtsbündig (Listing 3).

Die Prozedure FORMAT wird mit drei Parametern aufgerufen. EINGABE liest den zu formatierenden Text und schreibt ihn auf AUSGABE. RECHTS enthält die Spalte, in der jede Zeile enden soll. Für einen Testlauf können Sie RECHTS beispielsweise auf 40 Zeichen setzen.

Nachdem EINGABE und AUSGABE korrekt zum Lesen und Schreiben eröffnet wurden, wird jeweils eine Zeile mit READLINE eingelesen, entschieden, ob die Zeile formatiert werden muß und schließlich die formatierte Zeile ausgegeben. Beim Einlesen (READLINE) wird der Text in einen Zeilenpuffer LINE geschrieben. Alle Indizes in diesem

```
        I:=1;
        IF (N<=1) OR (S<=0) THEN (* 1:1 KOPIEREN *)
          WHILE I<=BIS DO
            BEGIN WRITE(AUSGABE, LINE[I]); I:= I+1 END
        ELSE
          BEGIN (* BERECHE ANZAHL DER LEERZEICHEN UND VERTEILUNG: *)
            IF ODD(ZEILE) THEN
              BEGIN                          (.* VON LINKS EINFÜGEN *)
                P:= S DIV (N-1);
                Q:= P+1;
                T:= P * (N-1) + N - S
              END
            ELSE
              BEGIN                          (* VON RECHTS EINFÜGEN *)
                Q:= S DIV (N-1);
                P:= Q+1;
                T:= S + 1 - Q * (N-1)
              END;
            FOR K:= 1 TO N-1 DO              (* WORTE KOPIEREN UND ERWEITERN *)
              BEGIN
                COPYWORD;
                IF K>=T THEN INSERTSPACES(Q)
                        ELSE INSERTSPACES(P)
              END;
            COPYWORD;                        (* LETZTES WORT *)
          END; (* IF *)
        WRITELN(AUSGABE);
      END; (* WHILE NOT EOF... *)
  END; (* FORMAT *)

BEGIN
  FORMAT(INPUT, OUTPUT, 40);
END.
```

Listing 3. Kopieren und Formatieren eines Textfiles (Schluß)

ARRAY von Zeichen sind als Ausschnittstypen

`TYPE LINEINDEX = 1..136`

deklariert. Außerdem bestimmt READLINE die Anzahl der Worte in der Zeile und die letzte Spalte, in der ein Buchstabe stand. Diese Werte kommen als Variablenparameter zurück. Anschließend wird die Anzahl der Leerzeichen bestimmt, die in die Zeile einzufügen sind, damit das letzte Zeichen in Spalte RECHTS erscheint. Enthält die Zeile nur ein Wort oder fehlen keine Leerzeichen, so kann der Zeilenpuffer LINE ohne Änderung ausgegeben werden.

Ansonsten beginnt die eigentliche Formatierung. Dabei sollten Sie vermeiden, daß sich die Worte alle am rechten und linken Rand sammeln. Daher werden in Zeilen mit ungerader Zeilennummer (»ODD(ZEILE)=TRUE«) Leerzeichen von links eingefügt, sonst jedoch von rechts.

Ein Beispiel: Steht in einer Zeile 11 Mal das Zeichen A mit einem Zwischenraum und ist RECHTS=40, so müssen 18 Leerzeichen auf 10 Wortzwischenräume verteilt werden. Ein Teil der Zwischenräume wird also um ein Leerzeichen, der Rest um zwei Leerzeichen erweitert. Im Programm übernehmen diese Verteilung die Variablen P, Q und T. Die ersten Worte werden mit P Leerstellen am Ende erweitert, während ab dem T-ten Wort Q Leerstellen angefügt werden. In der FOR-Schleife enthält also K die Nummer des gedruckten Wortes.

Nach diesen Ausführungen kennen Sie die Logik des Programmes, so daß Sie noch ein paar technische Details zum Thema Textfiles aufnehmen können: Zunächst erkennen Sie am Aufruf »FORMAT(INPUT,OUTPUT,40)« den Nutzen der vordefinierten Textfiles INPUT und OUTPUT, die ja auch im Programmkopf angegeben werden. Mit ihnen kann man die Tastatur und den Bildschirm wie ein normales File ansprechen. Bei der Prozedur READLINE wird die Funktion EOLN(Eingabe) zum Erkennen des Zeilenendes benutzt. Damit nach der Bearbeitung der ersten Zeile beim nächsten Aufruf die folgende Zeile weiterbearbeitet wird, muß am Ende der Aufruf READLN(EINGABE) stehen. Die Prozedur INSERTSPACES verwendet die Angabe eines Formatierungsparameters nach dem Doppelpunkt, um eine definierte Anzahl von Leerzeichen zu drucken:

`WRITE(AUSGABE, ' ': N)`

druckt ein Leerzeichen rechtsbündig in einem Feld der Größe N. Somit werden insgesamt N Leerzeichen ausgegeben. Schließlich erfolgt nach der Ausgabe jeder Zeile der Aufruf WRITELN(AUSGABE). Damit wird auf dem File AUSGABE ein Zeilenwechsel erzeugt, um die Zeilenstruktur des Files EINGABE zu erhalten.

Damit ist die Diskussion des Datentyps File und speziell der Textfiles beendet. Als Übung können Sie versuchen, das Programm in Listing 3 so zu verändern, daß es jede Zeile zentriert druckt. Sie können also die Prozedur READLINE wieder verwenden, jedoch dafür sorgen, daß die fehlenden Leerstellen zur Hälfte vor dem ersten Wort gedruckt werden.

(Florian Matthes/ev)



Von Zeigern, Listen und Graphen (Pascal, Teil 4)

Der letzte Beitrag unserer Einführung in die Programmiersprache Pascal beschäftigt sich mit einem nicht ganz einfachen, aber sehr interessanten Thema: den dynamischen Datenstrukturen.

Die Beschreibung der dynamischen Datenstrukturen steht grundsätzlich am Ende jeder Einführung in Pascal. Dies liegt einerseits daran, daß es sich dabei um ein besonders leistungsfähiges Sprachelement handelt, andererseits möchte man den Anfänger erst zum Schluß mit einem völlig neuen Verfahren zur Behandlung von Variablen konfrontieren. Haben Sie also in den vorhergehenden Artikeln Ihre ersten Schritte in Pascal gemeistert, dürfen Sie an dieser Stelle nicht frustriert zur Überzeugung gelangen, daß Pascal viel zu kompliziert und unverständlich ist. Vielmehr sollten Sie, nachdem Sie das bisher Gelernte in eigenen Programmen verwendet haben, mit diesen Erfahrungen den letzten Teil zur Fortbildung nutzen. Sicherlich gibt es auch einige Leser, die bisher zwar in Pascal programmiert haben, dabei jedoch einen weiten Bogen um Pointervariablen geschlagen haben. Für diese beginnt jetzt wohl der eigentlich interessante Teil der Artikelserie.

Alle bisher behandelten Datentypen und Variablen waren statisch. Am Beginn jedes Blockes wurden die lokalen Variablen angelegt und waren über ihren Namen veränderlich, bis der Block wieder verlassen und das Ende der Gültigkeit der Variablen erreicht wurde. Diese Variablenverwaltung hat zur Folge, daß bereits zur Übersetzungszeit der Compiler eine Speicherplatzverwaltung durchführen kann. Außerdem entspricht jedem Variablennamen ein (eventuell zusammengesetzter) Wert, jedoch gibt es auch gravierende Nachteile. Jedes Array besitzt eine feste Größe (es gibt keinen variablen DIM-Befehl wie zum Beispiel in Basic), so daß man oft entweder ein zu kleines Array definiert oder unnötig Speicherplatz verschwendet.

Man möchte also zur Laufzeit des Programms dynamisch entscheiden, ob Speicherplatz für eine Variable anzulegen ist oder ob eine bestehende Variable gelöscht werden soll. Außerdem möchte man auf diese dynamisch erzeugten Variablen gezielt zugreifen. Andererseits will man nicht auf die Typüberprüfungen des Compilers verzichten. Die Lösung des Dilemmas besteht darin, Variablen nicht mehr durch Namen, sondern durch Zeiger zu identifizieren.

Betrachten wir ein konkretes Beispiel. Es soll eine Kundenliste gebildet werden. Von jedem Kunden wird Name und Kundennummer gespeichert. Da die Anzahl der Kunden (in der Zukunft) unbekannt ist, scheidet ein Array von Kundenrecords als Datenstruktur aus. Statt die Daten auf einem langsamen externen Datenspeicher als File abzulegen, wird eine Liste mit Zeigern gebildet (Bild 1). Bildlich gesprochen entspricht jeder Record einem Kasten, der alle Daten eines Kunden enthält. Um nun einen Record im Programm anzusprechen, benutzt man keinen Variablennamen, sondern einen Zeiger auf diesen Kasten.

```
TYPE KUNDENZEIGER = ↑ KUNDE;
     KUNDE = RECORD
               NAME: ARRAY[1..10]
               OF CHAR;
               KNUMMER: INTEGER;
               NAECHSTER: KUNDEN-
               ZEIGER;
             END;
 VAR KUNDE1, KUNDENEU, LETZTER:
 KUNDENZEIGER;
```

In Bild 1 zeigt also der Zeiger KUNDE1 auf den ersten Kundenrecord. Von dort führt ein weiterer Zeiger zum nächsten Kundenrecord und so weiter. Jeder Zeiger (pointer) ist an einen Typ gebunden. So kann also eine Variable vom Typ KUNDENZEIGER nur auf einen Record vom Typ KUNDE weisen. Die Liste in Bild 1 ist also über einen Zeiger (KUNDE1) zugänglich. Indem man den Zeigern von Record zu Record folgt, kann man nacheinander jeden Record in der Liste adressieren.

Die obige Variablendeklaration definierte jeweils nur Speicherplatz für Zeiger auf Kundenrecords, jedoch keine Records selbst. Dies geschieht erst während der Programmausführung mit der Standardprozedur NEW:

```
NEW(KUNDE1)
```

Damit reserviert man irgendwo im Hauptspeicher des Rechners Speicherplatz für eine Variable des Typs, auf den die Pointervariable KUNDE1 zeigt. Um diesen neu erzeugten Kundenrecord zu adressieren, wird gleichzeitig dem Zeiger KUNDE1 die Adresse dieses Records zugewiesen (Bild 2a). Jetzt kann man der so erzeugten Variablen Werte übergeben:

```
KUNDE1 ↑ .NAME:= 'MAIER     ';
KUNDE1 ↑ .KNUMMER:= 100;
```

Während KUNDE1 eine Zeigervariable (vom Typ KUNDENZEIGER) ist, bezeichnet KUNDE↑ eine Variable des Typs KUNDE. Indem man also den Pfeil hinter eine Zeigervariable stellt, erhält man die dynamische Variable, auf die die Pointervariable zeigt. Man bezeichnet deshalb den Pfeil auch als »Dereferenzier-Operator«.

Da KUNDE1↑ eine (dynamische) Recordvariable ist, folgen nach einem Punkt wie üblich die Feldnamen des Records. Damit erhält man den Zustand aus Bild 2b. Um einen weiteren Kunden in die Liste aufzunehmen, ist zunächst wieder Speicherplatz zu reservieren.

```
NEW(KUNDENEU)
```

Wie oben kann man jetzt diesen Record mit Werten füllen (2c):

```
KUNDENEU↑ .NAME:= 'MÜLLER    ';
KUNDENEU↑ .KNUMMER:= 200;
```

Schließlich soll KUNDENEU als Nachfolger von KUNDE1 eingetragen werden. Hierzu wird im Feld NAECH-

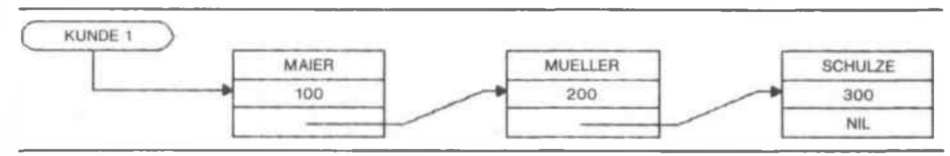


Bild 1. Eine Liste mit Zeigern



STER des Records, der durch KUNDE1 referiert wird, der Zeiger KUNDENEU eingetragen (Bild 2d).

```
KUNDE1↑ .NAECHSTER:= KUNDENEU
```

Um den letzten Record hinter KUNDENEU einzufügen, kann man folgende Anweisungsfolge verwenden:

```
NEW(LETZTER);
LETZTER↑ .NAME:='SCHULZE   ';
LETZTER↑ .KNUMMER:=300;
KUNDENEU↑ .NAECHSTER:= LETZTER
```

Damit ergibt sich eine Liste wie in Bild 2e. Wie erkennt man nun aber das Ende der Liste? Man muß wissen, ob das Feld NAECHSTER einen gültigen Zeiger enthält. Um anzuzeigen, daß ein Zeiger auf keine dynamische Variable weist, verwendet man den Wert NIL. Diese Konstante darf jeder Zeigervariablen zugewiesen werden. Mit

```
LETZTER↑.NAECHSTER:=NIL
```

gibt man also an, daß nach LETZTER↑ in der Liste kein Record mehr folgt.

Bisher programmierten wir alle Einfügungen in die Liste »zu Fuß«. Ein komplettes Programm zur Verwaltung einer Kundenliste zeigt Listing 1. Es speichert die Kunden in alphabetischer Reihenfolge. Zusätzlich existieren am Anfang und Ende der Liste je ein leerer Record. Damit ergibt sich eine Listenstruktur wie in Bild 3. Die Zeiger KOPF und ENDE weisen immer auf die beiden leeren Records. Im folgenden werden alle Funktionen des Programms anhand von Abbildungen erklärt.

Am einfachsten ist die Ausgabe der Tabelle (Bild 4a), siehe Prozedur TABELLE in Listing 1: Man durchläuft mit dem Zeiger Z die gesamte Liste und zeigt den jeweiligen Kundenrecord an. Mit

```
Z:= KOPF↑ .NAECHSTER
```

wird zunächst der leere (schraffierte) Record am Listenanfang übersprungen. Solange der Zeiger Z nicht mit dem Zeiger ENDE übereinstimmt, wird der Record Z↑ (nicht der Zeiger Z!) angezeigt. »Z:= Z↑ .NAECHSTER« führt schließlich von jedem Record zu seinem Nachfolger in der Liste.

Die Prozedur EINGABE liest zunächst von der Tastatur einen Namen ein. Dann wird durch den Aufruf der Prozedur VORHANDEN geprüft, ob dieser Name bereits in der Liste steht. Ist dies der Fall, so endet die Eingabe. Ansonsten wird dann ein neuer Record NEU geschaffen und mit Namen und Kundennummer gefüllt (Bild 4b). Da die Liste alphabetisch sortiert bleiben soll, muß NEU direkt hinter dem alphabetischen Vorgänger eingehängt werden. Daher liefert die Prozedur VORHANDEN einen Zeiger VOR, der in jedem Fall auf den Vorgänger in der Liste zeigt. Die Einfügung selbst geschieht dann in zwei Schritten. Zunächst wird der Zeiger NAECHSTER im neuen Record auf

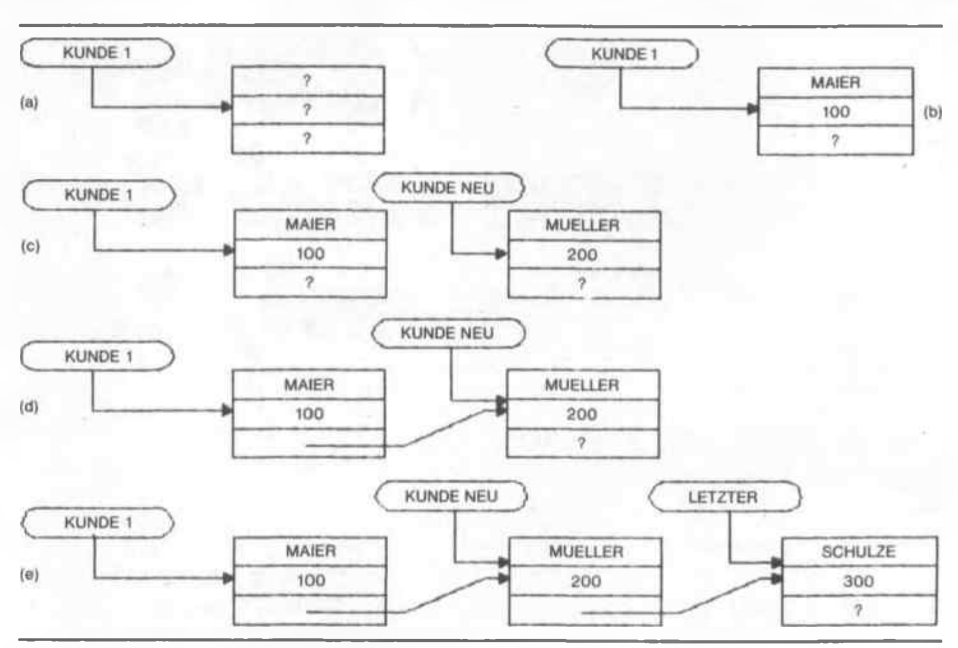


Bild 2. Operationen in der Liste (siehe Text)

```
PROGRAM KUNDENLISTE (INPUT, OUTPUT);
(* BEISPIEL FÜR DIE VERWALTUNG EINER LISTE MIT DYNAMISCHEN VARIABLEN *)
(* DIE DATEN VOM TYP KUNDE WERDEN STÄNDIG SORTIERT IN EINER EINFACH  *)
(* VERKETTETEN LISTE GESPEICHERT. JEDER RECORD BESITZT DAZU EINEN    *)
(* ZEIGER AUF DEN ALPHABETISCHEN NACHFOLGER. UM DAS EINFÜGEN UND     *)
(* LÖSCHEN EINFACH ZU GESTALTEN, BESITZT DIE LISTE JE EINEN LEEREN   *)
(* RECORD AM ANFANG UND ENDE.                                         *)

  CONST LEN = 10;            (* LÄNGE EINES NAMENS IN ZEICHEN        *)

  TYPE  STRING = ARRAY [1..LEN] OF CHAR;
        KUNDENZEIGER = ^ KUNDE;
        KUNDE  = RECORD
                   NAME     : STRING;
                   KNUMMER  : INTEGER;
                   NAECHSTER: KUNDENZEIGER;
                 END;
VAR  KOPF: KUNDENZEIGER; (* ZEIGER AUF DEN ERSTEN RECORD IN DER LISTE *)
     ENDE: KUNDENZEIGER; (* ZEIGER AUF DEN LETZTEN RECORD             *)
     CH  : CHAR;         (* BENUTZEREINGABE                           *)

PROCEDURE READSTRING (VAR S: STRING);
(* STRING MIT LEN ZEICHEN VON DER TASTATUR LESEN *)
  VAR I: INTEGER;
      C: CHAR;
BEGIN
  REPEAT READ(C) UNTIL C<>' '; (* VORLAUFENDE LEERZEICHEN IGNORIEREN  *)
  I:=1;

  REPEAT                       (* LEN ZEICHEN ODER BIS ZUM ZEILENENDE *)
    S[I]:= C; I:= I+1;
    READ(C)
  UNTIL (I>LEN) OR EOLN;

  WHILE I<LEN DO               (* S MIT LEERZEICHEN AUF DIE VOLLE     *)
    BEGIN                      (* LÄNGE ERWEITERN                     *)
      S[I]:= ' '; I:= I+1;
    END;
  READLN
END; (* READSTRING *)
FUNCTION VORHANDEN (S: STRING; VAR Z: KUNDENZEIGER): BOOLEAN;
(* SUCHT DEN NAMEN S IN DER LISTE. ERGEBNIS = TRUE, FALLS S GEFUNDEN *)
(* WURDE. Z ZEIGT BEI DER RÜCKKEHR IMMER AUF DIE POSITION DES ALPA-  *)
(* BETISCHEN VORGÄNGERS.                                              *)

   VAR Z1: KUNDENZEIGER;  (* Z1 STEHT IMMER EINEN RECORD WEITER ALS  *)
                          (* DER ZEIGER Z                            *)
BEGIN
  Z:= KOPF; Z1:= KOPF^.NAECHSTER; (* Z1 ZEIGT AUF ERSTEN GÜLTIGEN    *)
                                  (* RECORD IN DER LISTE             *)
  ENDE^.NAME:=S;                  (* MARKE IM LEZTEN RECORD DER LISTE*)

  WHILE Z1^.NAME<S DO             (* WANDERE MIT Z1 DURCH DIE LISTE, *)
    BEGIN                         (* BIS POSITION VON S ERREICHT     *)
      Z:= Z1; Z1:= Z1^.NAECHSTER
    END;
  (* GEFUNDEN, FALLS NAMEN GLEICH UND NICHT MARKE ERREICHT          *)
  VORHANDEN:= (Z1^.NAME=S) AND (Z1<>ENDE);
END; (* VORHANDEN *)

PROCEDURE DRUCKE (Z: KUNDENZEIGER);
```

Listing 1. Kundenverwaltung mit Listenstruktur

```
  BEGIN
    WITH Z^ DO
      WRITELN('NAME:', NAME: LEN+2, '   NUMMER:', KNUMMER :5)
  END; (* DRUCKE *)

  PROCEDURE EINGABE;
  (* EINFÜGEN EINES NEUEN KUNDENRECORDS AN DER KORREKTEN POSITION IN    *)
  (* DER KUNDENLISTE ZWISCHEN VON UND BIS                                *)
    VAR N  : STRING;        (* NEUER NAME                                *)
        NEU: KUNDENZEIGER; (* ZEIGER AUF DEN NEUEN RECORD                *)
        VOR: KUNDENZEIGER; (* ZEIGER AUF DEN VORGÄNGER IN DER LISTE      *)

  BEGIN
    WRITE('NAME:'); READSTRING(N);
    IF VORHANDEN (N,VOR) THEN     (* VOR WIRD HIER GESETZT!             *)
      WRITELN(N, ' IST BEREITS KUNDE!')
    ELSE
      BEGIN
        NEW (NEU);                 (* NEUEN RECORD BESORGEN              *)
        WRITE ('KUNDENNUMMER:');
        READLN (NEU^.KNUMMER);     (* UND MIT WERTEN BELEGEN:            *)
        NEU^.NAME:= N;

        (* NEU^ HINTER VOR^ EINFÜGEN: *)
        NEU^.NAECHSTER:= VOR^.NAECHSTER;
        VOR^.NAECHSTER:= NEU
      END;
  END; (* EINGABE *)

  PROCEDURE AUSGABE;
  (* AUSGABE EINES KUNDENRECORDS                                         *)
    VAR N  : STRING;
        VOR: KUNDENZEIGER;  (* ZEIGER AUF ALPHABETISCHEN VORGÄNGER       *)
  BEGIN
    WRITE ('NAME:');
    READSTRING(N);
    IF VORHANDEN (N,VOR) THEN   (* VOR WIRD AUF DEN VORGÄNGER GESETZT! *)
      DRUCKE (VOR^.NAECHSTER)
    ELSE
      WRITELN (N, ' NICHT ALS KUNDE GESPEICHERT!');
  END; (* AUSGABE *)

  PROCEDURE LOESCHEN;
  (* LOESCHEN EINES KUNDENRECORDS IN DER LISTE                          *)
    VAR N  : STRING;
        VOR: KUNDENZEIGER;  (* WIEDERUM ZEIGER AUF DEN VORGÄNGER IN    *)
                            (* DER LISTE DER KUNDEN                     *)

  BEGIN
    WRITE('NAME:'); READSTRING(N);
    IF VORHANDEN(N,VOR) THEN  (* VOR WIRD AUF DEN VORGÄNGER GESETZT     *)
      BEGIN
        WRITELN ('GELÖSCHT WURDE:');
        DRUCKE (VOR^.NAECHSTER); (* ZUR SICHERHEIT ANZEIGEN             *)

        (* JETZT DEN NACHFOLGER VON VOR^ AUS DER LISTE ENTFERNEN:        *)
        VOR^.NAECHSTER:= VOR^.NAECHSTER^.NAECHSTER;
      END
    ELSE
      WRITELN (N, ' NICHT ALS KUNDE GESPEICHERT!');
  END; (* LOESCHEN *)

  PROCEDURE TABELLE;
  (* DRUCKE EINE ALPHABETISCHE AUFLISTUNG ALLER KUNDEN *)
    VAR Z: KUNDENZEIGER;
  BEGIN
    Z:= KOPF^.NAECHSTER;         (* Z AUF DEN ERSTEN BELEGTEN RECORD    *)
    WHILE Z<>ENDE DO             (* SOLANGE NICHT DER LETZTE (LEERE)    *)
      BEGIN                      (* RECORD ERREICHT WIRD:               *)
        DRUCKE(Z);               (* ANZEIGE Z^                          *)
        Z:= Z^.NAECHSTER;        (* ZUM NÄCHSTEN KUNDEN                 *)
      END;
  END; (* TABELLE *)

BEGIN (* HAUPTPROGRAMM *)
  NEW(KOPF); NEW(ENDE);          (* ERSTEN UND LETZTEN RECORD BILDEN    *)
  KOPF^.NAECHSTER:= ENDE;        (* ENDE DIREKT NACH DEM KOPF, ALSO IST *)
                                 (* DIE LISTE LEER                      *)

  REPEAT
    WRITELN('E INGABE     L OESCHEN');
    WRITELN('A USGABE     T ABELLE');
    WRITELN('      X   BEENDEN');
    READLN(CH);
    IF CH IN ['E','A','L','T','X'] THEN
      CASE CH OF
        'E': EINGABE;
        'A': AUSGABE;
        'L': LOESCHEN;
        'T': TABELLE;
        'X': (* NICHTS *)
      END (* CASE *)
    ELSE
      WRITELN(CH ,' IST NICHT MÖGLICH!');
  UNTIL CH='X'
END.
```

Listing 1. Kundenverwaltung mit Listenstruktur (Schluß). Das Zeichen »^« entspricht dem Hochpfeil (»↑«)

den Nachfolger des Records VOR↑ gesetzt. Dann kann der Zeiger NAECHSTER in VOR↑ auf den Wert des Zeigers NEU gesetzt werden. Damit ergibt sich eine Liste wie Bild 4c. Dort sind die geänderten Zeiger fett gezeichnet.

Um in der Prozedur AUSGABE zu einem Namen den zugehörigen Kundenrecord in der Liste zu finden, dient ebenfalls die Funktion VORHANDEN. Da sie jedoch immer einen Zeiger auf den Vorgänger liefert, muß der Record VOR↑.NAECHSTER verwendet werden.

Ebenso einfach ist das Löschen eines Kunden. In der Prozedur LOESCHEN wird wieder mit VORHANDEN ein Zeiger auf den Vorgänger in der Liste gesetzt. Mit

`VOR↑.NAECHSTER:=VOR↑.NAECHSTER↑.NAECHSTER`

wird im Record VOR↑ der Zeiger auf den übernächsten Nachfolger gerichtet (fette Linie in Bild 4d).

Jetzt ist es an der Zeit, uns mit der Funktion VORHANDEN näher zu befassen. Sie durchsucht die alphabetisch sortierte Liste nach dem Namen S. Wird ein Record mit NAME=S gefunden, so ist VORHANDEN gleich TRUE. Der Zeiger Z weist dann auf den Vorgänger dieses Records. Ist jedoch VORHANDEN gleich FALSE, so existiert kein Kunde mit dem Namen S. Jetzt zeigt Z auf den Record, hinter dem ein neuer Record eingefügt werden müßte, um die alphabetische Ordnung zu erhalten. In Bild 4e ist gezeigt, daß zwei Zeiger Z und Z1 verwendet werden. Man durchläuft wie bei der Prozedur TABELLE die Liste bis gilt:

`Z1↑.NAME >=S`

Dabei hinkt der Zeiger Z immer einen Record hinter dem Zeiger Z1 her. Ist die obige Bedingung eingetreten, so zeigt Z auf den alphabetischen Vorgänger. Um zu verhindern, über das Listenende hinauszulaufen, wird am Anfang der letzte (unbenutzte) Record mit dem gesuchten Namen gefüllt:

`ENDE↑.NAME:=S; (z.B. S='SCHULZE')`

Damit bricht die WHILE-Schleife sicher für Z1=ENDE ab. Einen solchen Eintrag zur Vereinfachung des Abbruchkriteriums bezeichnet man als Marke. Das boolesche Ergebnis der Funktion bestimmt also abschließend die folgende Zuweisung:

`VORHANDEN:= (Z1↑.NAME=S) AND (Z1<> ENDE)`

Am Programmanfang wird mit

`NEW(KOPF); NEW(ENDE); KOPF↑.NAECHSTER:= ENDE`

eine Liste mit zwei unbenutzten Records erzeugt. Durch diese Initialisierung findet Einfügen und Löschen immer zwischen zwei Records statt. Wären diese Hilfsrecords nicht vorhan-

den, so müßte man zum Beispiel beim Löschen immer die Sonderfälle Löschen am Listenanfang oder Löschen des einzigen Elements in der Liste prüfen, da in diesen Fällen auch die Zeiger KOPF und ENDE verändert werden müßten.

Jetzt sollten Sie sich zunächst etwas Zeit nehmen, alle diese Informationen zu verdauen und sich genauer mit dem Listing 1 auseinandersetzen. Dann führen Sie einen kleinen Verständnistest durch:

1. Was ist der Unterschied zwischen den folgenden Zuweisungen?

```
KOPF  := ENDE            und
KOPF↑ := ENDE↑?
```

2. Ändern Sie das Programm so, daß die Namen in umgekehrter alphabetischer Reihenfolge (SCHULZE, MUELLER, MAIER) in der Liste gespeichert werden!

Zur Beantwortung der ersten Frage blättern Sie bei Bedarf zum Anfang der Ausführungen über Zweigvariablen zurück. Die zweite Aufgabe besteht »nur« in der Änderung der Funktion VORHANDEN.

Es gibt sehr viele verschiedene Varianten mit dem Konzept der Speicherung von Daten in Listen. Sie unterscheiden sich in der Methode, wie die Records durch Zeiger verkettet sind. Im Beispiel der Kundenliste kann man ohne Probleme von jedem Kunden zu seinem alphabetischen Nachfolger gelangen. Jedoch erreicht man den Vorgänger im Alphabet nur, indem man die Liste vom Kopf her durchläuft. Eine naheliegende Lösung besteht darin, jeden Record um einen Zeiger auf den Vorgänger zu erweitern:

```
TYPE KUNDENZEIGER =  KUNDE;
     KUNDE = RECORD
       NAME : ARRAY
              [1..10] OF CHAR;
       KNUMMER  :  INTEGER;
       VORIGER  :  KUNDENZEIGER;
       NAECHSTER:  KUNDENZEIGER;
     END;
```

Diese Verkettung erlaubt zwar ein Durchlaufen der Liste vorwärts wie rückwärts, jedoch werden die Operationen zum Einfügen und Löschen wesentlich komplexer, da sie zwei Verkettungen aktualisieren müssen. Eine solche doppelt verkettete Liste zeigt symbolisch das Bild 5.

Es gibt noch viele andere Methoden, Datenstrukturen mit Zeigern zu bilden (zum Beispiel Bäume), in denen man sehr effizient Werte sortiert einfügen, suchen und löschen kann. Da dieses Themengebiet sehr umfangreich ist und wirklich gute Literatur über dieses Thema existiert [1], sollen die letzten Beispiele mit Zeigern andere Anwendungen zeigen.

Zunächst wird ein Verfahren vorgestellt, das Werte so speichert, daß man ohne Suchen in einem Schritt einen vorgegebenen Wert wiederfindet. Die angenommene Aufgabe besteht darin, für eine Anzahl von Orten die Postleitzahl abzulegen. Man soll also einerseits neue Orte mit ihrer Postleitzahl eingeben können und andererseits zu einem vorgegebenen Namen die gespeicherte Postleitzahl (falls gespeichert) erhalten.

Bei der Speicherung mit dynamischen Variablen im Programm KUNDENLISTE trat das Problem auf, daß man sich erst mit einer (linearen) Suche durch die Liste bewegen muß, um einen Kundenrecord zu finden. Dabei muß man bei n gespeicherten Werten im Durchschnitt n/2 Vergleiche aufwenden. Optimal wäre eine Speicherungsmethode, bei der man direkt aus dem Suchschlüssel einen Verweis auf die gespeicherten Informationen erhält. Diese Anforderung erfüllt das sogenannte »Hashing«: Man speichert alle Daten in einer Hash-Tabelle. Das ist ein Array, das mit ganzen Zahlen indiziert wird:

```
CONST HASHSIZE = 97;
TYPE INFO = INTEGER; (* POSTLEIT-
ZAHL *)
VAR HASHTAB  = ARRAY[0..HASH-
               SIZE] OF INFO;
```

Nun sollen im vorliegenden Beispiel als Schlüssel zwanzigstellige Städtenamen verwendet werden. Man steht also vor dem Problem, aus einem String der Länge 20 einen eindeutigen Index zwischen 0 und 1000 zu erzeugen. Eine solche Hash-Funktion – Schlüsseltransformations-Funktion – gibt das Programm in Listing 2 unter dem Namen HASHINDEX wieder. Zunächst wird die Summe aller Codezahlen der Zeichen berechnet:

```
F = 70
R = 82
A = 65
N = 78
K = 75
F = 70
U = 85
R = 82
T = 84
_ = 32
_ = 32
...
_ = 32
------
  1043
```

Anschließend wird dieser Index durch die Tabellengröße geteilt. Der Divisionsrest ist eine Zahl zwischen 0 und der Tabellengröße und kann somit direkt als Index gelten.

```
1034 / 97 = 10 REST 73
```

Um die Postleitzahl für Frankfurt (6000) zu speichern, sind folgende Schritte erforderlich:

```
INDEX:=HASHINDEX('FRANKFURT    ');
HASHTAB[INDEX] := 6000
```

Index besitzt also den Wert 73. Genauso einfach ist es, die Postleitzahl für Frankfurt anzuzeigen:

```
INDEX:=HASHINDEX('FRANKFURT    ');
WRITELN('PLZ:', HASHTAB[INDEX];
```

Das Verfahren besitzt jedoch einen gravierenden Nachteil. Die Hashfunktion, die zu einem Städtenamen einen Index zwischen 0 und HASHSIZE bestimmt, muß nicht eindeutig sein. Es kann also durchaus sein, daß es einen weiteren Städtenamen (sagen wir X-Stadt) gibt, der bei der obigen Berechnung ebenfalls den Index 73 ergibt. Dann kommt es in der Hashtabelle zu einer Kollision (»hash clash«). In diesem Fall ist es mit der obigen Datenstruktur unmöglich festzustellen, ob in HASHTAB[73] die Postleitzahl von Frankfurt oder X-Stadt steht. Im Programm nach Listing 2 wird das bisherige Konzept deshalb folgendermaßen modifiziert:

```
TYPE HASHELEMENT = RECORD
               NAME: STRING;
               PLZ : INTEGER;
               NEXT:↑HASHELEMENT
             END;
VAR HASHTAB: ARRAY[0..HASHSIZE] OF
               ↑HASHELEMENT;
```

Man speichert in der Hashtabelle nur einen Zeiger auf einen Record vom Typ Hashelement. Dieses Record enthält neben der eigentlichen Information (PLZ) noch den Name der Stadt. Ein Record vom Typ HASHELEMENT ist natürlich nur dann erforderlich, falls eine Postleitzahl gespeichert werden soll. Ansonsten besitzt der Zeiger in HASHTAB den Wert NIL. Diese Vorbelegung führt am Programmanfang die Prozedur LEERETABELLE durch. Das Feld NEXT im Record HASHELEMENT dient zur Behandlung von Kollisionen. Alle Schlüssel, die unter demselben Index stehen, werden in zufälliger Reihenfolge zu einer Liste mit dem Zeiger NEXT verkettet. Beim Einfügen und beim Abfragen muß deshalb eventuell diese Liste durchlaufen werden. Bild 6 zeigt die Datenstruktur der Hashtabelle. Jeder Zeiger im Array HASHTAB kann also Kopf einer Liste von Einträgen sein. Bitte nehmen Sie die Werte in der Abbildung nicht zu genau, da aus naheliegenden Gründen nicht für jeden Namen die Hashfunktion berechnet wurde.

Unter folgenden Bedingungen ist Hashing die mit Abstand beste Speicherungsform und jeder anderen Speichermethode in Geschwindigkeit und Programmieraufwand weit überlegen:

1. Es existiert eine Grenze für die Anzahl der Werte, die gespeichert werden sollen. Die Größe der Hashtabelle ist etwa um zehn Prozent größer als diese Maximalanzahl festzulegen.

2. Eine sortierte Ausgabe in der Reihenfolge der Schlüssel ist nicht erforderlich. Es werden nämlich die Schlüssel im Idealfall völlig ungeordnet über die Hash-Tabelle verteilt, so daß keine Möglichkeit existiert, von einem Eintrag zu seinem Nachfolger zu gelangen.

Aus dem bisher Gesagten ist klar, daß die Hash-Funktion alle Schlüssel (im Beispiel die Städtenamen) möglichst gleichmäßig über den gesamten Indexbereich verteilen soll. Dabei muß man natürlich vermeiden, daß häufig auftretende ähnliche Schlüssel denselben Index erhalten. Soll zum Beispiel die Häufigkeit von Variablennamen in Pascal bestimmt werden, so darf eine Hashfunktion keinesfalls alle Namen mit nur einem Buchstaben auf denselben Index abbilden, da sonst ständig eine lange Liste durchsucht werden muß.

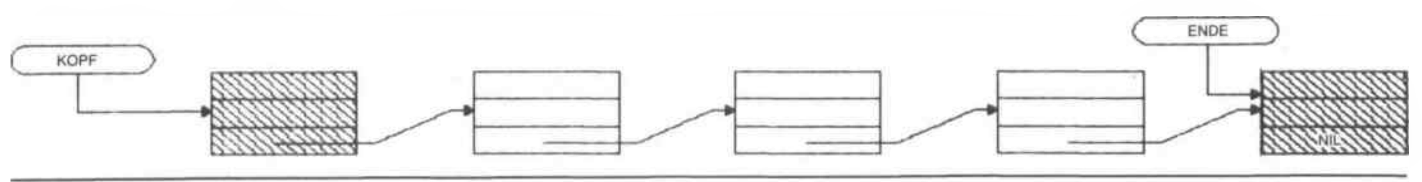


Bild 3. Kundenliste mit leerem Kopf und Ende

a) Durchlaufen der Liste
Z
ENDE
KOPF
NIL

b) Vor dem Einfügen
NEU
VOR
ENDE
KOPF
NIL

c) Nach dem Einfügen
NEU
VOR
ENDE
KOPF
NIL

d) Löschen in der Liste
VOR
ENDE
KOPF
NIL

e) Eine Marke am Listenende
Z
Z1
ENDE
KOPF
SCHULZE
?
NIL

f) leere Liste am Programmanfang
ENDE
KOPF
NIL

Bild 4. Die Kundenliste aus Listing 1

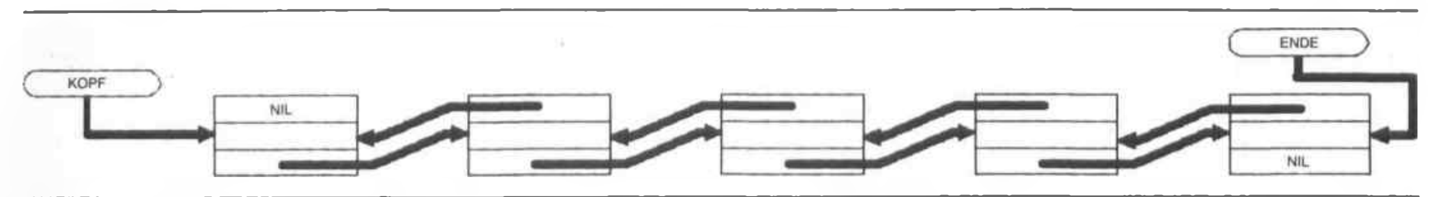


Bild 5. Eine doppelt verkettete Liste

Normalerweise ist die Hash-Funktion jedoch unkritisch, so daß man jede Information mit nur einem Zugriff auf einen Record erhält. Bemerkenswert ist noch die Speicherplatzökonomie des Verfahrens. Zwar ist ständig ein Array der Größe HASHSIZE vorhanden, dieses enthält jedoch nur Zeiger (zwei Byte Länge). Nur für tatsächlich vorhandene Schlüssel wird ein Record vom Typ HASHELEMENT dynamisch erzeugt, der neben der eigentlichen Information nur noch Speicherplatz für einen Zeiger (NEXT) benötigt. Unter den oben genannten zwei Bedingungen ist Hashing also wärmstens zu empfehlen, da es die Geschwindigkeit von Array-Zugriffen mit der Effizienz dynamischer Variablen verbindet.

Als abschließendes Beispiel steht an, eine Aufgabe mit Graphen in Pascal zu lösen. Ein Graph ist in der Mathematik ein abstraktes Gebilde aus Knoten und Kanten, die von Knoten zu Knoten führen. Uns interessiert eine spezielle Art von Graphen, nämlich sloche, deren Kanten gerichtet und markiert sind. Das klingt schrecklich abstrakt, läßt sich jedoch leicht mit einer Abbildung (Bild 7) erklären: Ein Knoten wird durch einen Kreis mit der Nummer des Knotens dargestellt, während ein Pfeil mit einem Zeichen zwischen zwei Knoten eine gerichtete, markierte Kante symbolisiert. Von Knoten 1 führt also eine Kante nach 2, die mit X markiert ist.

Nachdem nun die grundlegenden Begriffe bekannt sind, zur eigentlichen Aufgabe. Durch einen Graphen kann man einfache »Sprachen« beschreiben: Bild 8 zeigt einen Graphen, der alle Wörter der Form

```
AB
ABAB
ABABAB
ABABABAB..
```

»erkennt«. Dies geschieht folgendermaßen: Am Anfang setzt man eine »Marke« auf den Eingangsknoten 1. Außerdem gibt man das Wort vor, von dem man wissen will, ob es vom Graph erkannt wird:

```
ABAB
```

Nun liest man Buchstabe für Buchstabe und bewegt die Marke entsprechend den Zeichen an den Kanten durch den Graphen. Mit dem ersten Buchstaben des Wortes »ABAB« erreicht man von Knoten 1 den Knoten 2, da die Kante von 1 nach 2 mit dem Zeichen »A« markiert ist. Da der zweite Buchstabe ein »B« ist, wandert die Marke vom Knoten 2 entlang der Kante mit der Markierung »B« zum Knoten 3. Im dritten Schritt wird der dritte Buchstabe (wieder ein »A«) untersucht, so daß sich die Marke von Knoten 3 zurück zu Knoten 2 bewegt. Mit dem letzten Buchstaben »B« kommt die Marke

```
PROGRAM HASHING (INPUT, OUTPUT);
(* SPEICHERUNG VON POSTLEITZAHLEN MIT HASHING UND ÜBERLAUFLISTEN *)

  CONST HASHSIZE = 97; (* PRIMZAHL *)
        LEN = 20;

  TYPE  STRING = ARRAY [1..LEN] OF CHAR;
        HASHELEMENT = RECORD
                        NAME : STRING;
                        PLZ  : INTEGER;
                        NEXT : ^ HASHELEMENT
                      END;

  VAR   HASHTAB  : ARRAY[0..HASHSIZE] OF ^ HASHELEMENT;
        N        : STRING;
        PLZ      : INTEGER;

  PROCEDURE READSTRING (VAR S: STRING);
  (* STRING MIT LEN ZEICHEN VON DER TASTATUR LESEN *)
    VAR I: INTEGER;
        C: CHAR;
  BEGIN
    REPEAT READ(C) UNTIL C<>' '; (* VORLAUFENDE LEERZEICHEN IGNORIEREN  *)
    I:=1;

    REPEAT                      (* LEN ZEICHEN ODER BIS ZUM ZEILENENDE *)
      S[I]:= C; I:= I+1;
      READ(C)
    UNTIL (I>LEN) OR EOLN;

    WHILE I<LEN DO              (* S MIT LEERZEICHEN AUF DIE VOLLE     *)
      BEGIN                     (* LÄNGE ERWEITERN                     *)
        S[I]:= ' '; I:= I+1;
      END;
    READLN
  END; (* READSTRING *)

  FUNCTION HASHINDEX (NAME: STRING): INTEGER;
  (* LIEFERT DEN INDEX IN DER HASHTABELLE FÜR DIESEN NAMEN *)
    VAR I    : 1..LEN;
        INDEX: INTEGER;
  BEGIN
    INDEX:= 0;
    FOR I:= 1 TO LEN DO INDEX:= INDEX + ORD(NAME[I]);
    HASHINDEX:= INDEX MOD HASHSIZE;
  END; (* HASHINDEX *)
PROCEDURE SPEICHERE (N: STRING; POSTLEITZAHL: INTEGER);
(* EINTRAG NAME MIT POSTLEITZAHL. ZUR SICHERHEIT TEST, OB DOPPELT *)

   VAR INDEX   : INTEGER;             (* INDEX IN HASHTAB            *)
       P,NEU   : ^HASHELEMENT;        (* ZEIGER IN ÜBERLAUFLISTE     *)
       DOPPELT: BOOLEAN;

BEGIN
  INDEX:= HASHINDEX(N);               (* INDEX DES NAMENS BERECHNEN  *)
  P:= HASHTAB[INDEX];                 (* ZEIGER AUS DER HASHTABELLE  *)

  DOPPELT:= FALSE;                    (* PRÜFE AUF DOPPELTEN EINTRAG, *)
  WHILE P<>NIL DO                     (* FALLS AN DIESER POSITION    *)
    BEGIN                             (* BEREITS EINTRÄGE EXISTIEREN. *)
      DOPPELT:= DOPPELT OR (P^.NAME=N);
      P:= P^.NEXT
    END;

 IF DOPPELT THEN
    WRITELN(N, ' IST BEREITS GESPEICHERT!')
  ELSE
    BEGIN
      NEW(NEU);                         (* NEUEN RECORD ERZEUGEN        *)
      WITH NEU^ DO
        BEGIN
          NAME:=N; PLZ:= POSTLEITZAHL;(* WERTE EINTRAGEN               *)
          NEXT:= HASHTAB[INDEX]         (* VOR DEN ALTEN WERTEN EINFÜGEN *)
        END;
      HASHTAB[INDEX]:= NEU              (* NEU STEHT AN 1. POSITION     *)
    END; (* IF *)
END; (* SPEICHERE *)

FUNCTION POSTLEITZAHL(NAME: STRING): INTEGER;
(* LIEFERT DIE POSTLEITZAHL ZU DIESEM NAMEN ODER DIE ZAHL 0 *)
  VAR P     : ^ HASHELEMENT;
      INDEX: INTEGER;
BEGIN
  INDEX:= HASHINDEX(NAME);              (* BESTIMME DEN INDEX IN HASHTAB *)
  POSTLEITZAHL:= 0;                     (* VORLÄUFIG NOCH NICHT GEFUNDEN *)

  P:= HASHTAB[INDEX];                   (* DURCHSUCHE DIE ÜBERLAUFLISTE  *)
  WHILE P<>NIL DO
    IF P^.NAME=NAME THEN
      BEGIN
        POSTLEITZAHL:= P^.PLZ;          (* FUNKTIONSERGEBNIS FESTLEGEN   *)
        P:= NIL                         (* ENDE DER SCHLEIFE ERZWINGEN   *)
      END
    ELSE
      P:= P^.NEXT;                      (* SONST WEITERSUCHEN            *)
 END; (* POSTLEITZAHL *)

 PROCEDURE LEERETABELLE;
```

Listing 2. Datenspeicherung mit Hashing

```
(* LÖSCHE DIE GESAMTE HASHTABELLE *)
  VAR I: 0..HASHSIZE;
BEGIN
  FOR I:= 0 TO HASHSIZE DO HASHTAB[I]:= NIL
END; (* LEERETABELLE *)
BEGIN (* HAUPTPROGRAMM *)
  LEERETABELLE;                              (* NOCH IST NICHTS GESPEICHERT     *)
  WRITELN('SPEICHERUNG VON POSTLEITZAHLEN: (BEENDEN MIT NAME = *)');

  WRITE('NAME:');READSTRING(N);
  WHILE N[1]<>'*' DO
    BEGIN
      WRITE('PLZ :'); READ(PLZ);
      IF PLZ = 0 THEN                        (* SUCHE DIE ZUGEHÖRIGE PLZ:       *)
        WRITELN(POSTLEITZAHL(N))
      ELSE
        BEGIN
          WRITELN;
          SPEICHERE(N,PLZ);                  (* SPEICHERE NAMEN MIT DIESER PLZ *)
        END;
      WRITE('NAME:'); READSTRING(N)
    END; (* WHILE *)
END.
```

Listing 2. Datenspeicherung mit Hashing (Schluß)

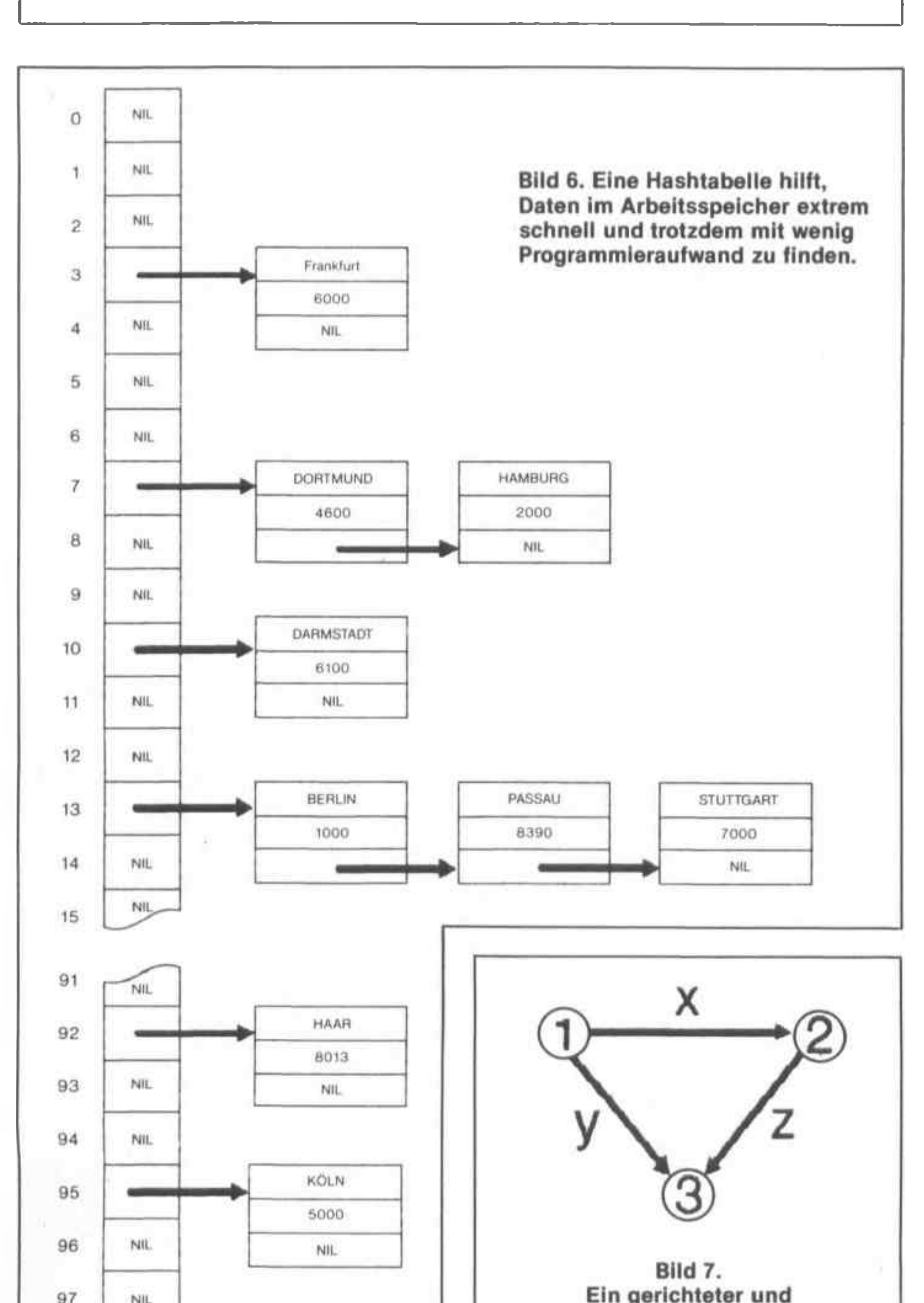


Bild 6. Eine Hashtabelle hilft, Daten im Arbeitsspeicher extrem schnell und trotzdem mit wenig Programmieraufwand zu finden.

Bild 7. Ein gerichteter und markierter Graph

schließlich auf Knoten 3 zu liegen. Da nun das Wort zu Ende ist, kann man aus der Position der Marke entscheiden, ob der Graph das Wort »erkannt« hat. Die Marke befindet sich im Knoten 3, den ein dicker Rand hervorhebt. Diese Knoten sind »akzeptierende« Knoten. Befindet sich die Marke am Schluß auf einem akzeptierenden Knoten, so ist das gesamte Wort erkannt und gehört somit zum Sprachschatz des Graphen. Sicher verstehen Sie jetzt auch, warum der Graph in Bild 8 alle Worte erkennt, die aus einer beliebig langen Folge von »AB« bestehen. Mit jedem »A« wird die Marke auf den Knoten 2 gesetzt, von wo aus mit »B« der akzeptierende Knoten 3 erreicht wird. Hätten wir jedoch das Wort

```
ABA
```

der obigen Testprozedur unterzogen, so wäre die Marke im dritten Schritt ebenfalls im Knoten 2 gelandet. Da das Wort an dieser Stelle bereits zu Ende ist, die Marke aber nicht auf einem akzeptierenden Knoten liegt, gehört »ABA« nicht zur Sprache des Graphen. Auch das Wort

```
ABAA
```

bleibt unerkannt, da mit ABA die Marke auf Knoten 2 liegt. Für den folgenden Buchstaben »A« existiert jedoch keine Markierung an einer Kante von 2 aus, so daß das gesamte Wort nicht erkannt wird.

Bild 9 zeigt einen Graphen, der alle Worte der Form

```
AAB
AABAAB
AABAABAAB
AABAABAABAAB...
```

identifiziert. Das können Sie bei der Anwendung der obigen Regeln für einige Beispielworte leicht herausfinden.

Listing 3 stellt ein Programm dar, das das mühsame Verfolgen des Weges der Marke durch den Graphen automatisch durchführt. Genauer gesagt soll das Programm folgendes leisten:

1. Es wird eine Beschreibung des Graphen eingelesen.
2. Es wird geprüft, ob überhaupt ein akzeptierender Knoten erreicht werden kann (siehe Bild 10 für ein Gegenbeispiel).
3. Für beliebig vorgegebene Wörter wird geprüft, ob diese erkannt werden.

Zunächst muß eine Datenstruktur für die interne Repräsentation des Graphen im Speicher des Computers gefunden werden. Da es beliebig viele Knoten und Kanten geben kann, wird man diese mit dynamischen Variablen darstellen. Graphen sind nämlich geradezu das klassische Beispiel für die Verwendung von Zeigern. Für jeden Knoten speichert man seine Nummer und eine Liste der Kanten, die von die-

sem Knoten wegführen. Jede Kante in dieser Liste enthält ihrerseits die Markierung und einen Zeiger auf den Knoten, zu dem sie führt.

```
TYPE TKNOTENREF =↑ TKNOTEN;
  TKANTENREF =↑ TKANTE;
  TKNOTEN     =  RECORD
        NUMMER: INTEGER;
        KANTENLISTE: TKANTENREF
       END;
  TKANTE      = RECORD
        MARKIERUNG : CHAR;
        NEXT: TKANTENREF;
        NACH: TKNOTENREF;
       END;
```

Es gibt also zwei Zeigertypen. Zeiger vom Typ TKNOTENREF zeigen immer auf Knoten (Records vom Typ TKNOTEN), während Zeiger vom Typ TKANTENREF immer auf Kanten (Records vom Typ TKANTE) weisen. Die Bilder 11 und 12 verdeutlichen jeweils die interne Darstellung eines Graphen. Die großen Kästen verbildlichen Records vom Typ TKNOTEN. Sie enthalten also die Nummer des Knotens. Diese Nummer ist negativ, falls es sich bei dem Knoten um einen akzeptierenden Knoten handelt. Außerdem ist jeder Knoten Kopf einer Liste von Records des Typs TKANTE (kleine Kästen). Für jede Kante wird das Zeichen, mit dem die Kante markiert ist, und ein Zeiger auf den Knoten am Ende der Kante, gespeichert. Da alle Kanten, die an einem Knoten beginnen, zu einer Liste verkettet sind, wird noch das Feld NEXT benötigt, in dem ein Zeiger auf die nächste Kante in der Liste enthalten ist. Von »außen« erreicht man den gesamten Graphen nur durch die Zeigervariable ANFANG.

Als erstes nun zur Funktion ERKANNT. Sie verwendet diese Datenstruktur, um zu prüfen, ob das Wort in dem Array W

```
VAR W: ARRAY[1..100] OF CHAR;
```

akzeptiert wird. Die Strategie ist sehr einfach und entspricht dem obigen Wandern einer Marke durch den Graphen. Der Parameter Q gibt die momentane Position der Marke (auf einem Knoten) an. I indiziert den momentan bearbeiteten Buchstaben im Wort W. Das Ende des Wortes markiert ein Dollar-Zeichen »$«. Wurde dieses Ende gelesen, so ist das Wort genau dann akzeptiert, wenn der momentane Knoten (Q↑) ein akzeptierender Knoten ist (Nummer ist negativ). Ansonsten wird die Kantenliste nach einer Kante mit der passenden Markierung durchsucht. Stimmt die Markierung mit dem laufenden Buchstaben W[I] überein, so setzt sich die Prüfung mit dem nächsten Buchstaben und dem Endknoten der Kante fort. Die Funktion ERKANNT ist also rekursiv. Bemerkenswert ist noch die Tatsache, daß von einem Knoten eventuell zwei oder mehrere gleich markierte Kanten

```
PROGRAM GRAPH (INPUT, OUTPUT);
(* DIESES PROGRAMM DEMONSTRIERT DIE VERWENDUNG VON DYNAMISCHEN VARIABLEN *)
(* ZUR DARSTELLUNG EINES GERICHTETEN, MARKIERTEN GRAPHEN IM SPEICHER DES *)
(* COMPUTERS. DIE PROZEDUR GRAPHEINLESEN BAUT EIN NETZWERK AUS RECORDS   *)
(* DER TYPEN TKNOTEN UND TKANTE AUF, DESSEN ERSTER KNOTEN ÜBER DEN       *)
(* ZEIGER ANFANG ERREICHT WERDEN KANN. DIE FUNKTIONEN ISTLEER UND        *)
(* ERKANNT BENUTZEN DIESE DATENSTRUKTUR, UM DIE DURCH DEN GRAPHEN BE-    *)
(* SCHRIEBENE SPRACHE NÄHER ZU UNTERSUCHEN (S. TEXT)                     *)

  CONST ENDE = '$';            (* MARKIERUNG AM ENDE DER EINGABE          *)

  TYPE  TEINGABE = CHAR;       (* DIE MARKIERUNGEN DER KANTEN BESTEHEN    *)
                               (* AUS EINZELNEN ZEICHEN                   *)
        TKNOTENREF = ^TKNOTEN; (* ZEIGER AUF EINEN KNOTEN                 *)
        TKANTENREF = ^TKANTE;  (* ZEIGER AUF EINE KANTE                   *)
        TKNOTEN = RECORD
                    NUMMER: INTEGER;    (* NEGATIV, FALLS AKZEPTIEREND    *)
                    ISTMARKIERT: BOOLEAN; (* FÜR FUNKTION ISTLEER         *)
                    KANTENLISTE: TKANTENREF; (* LISTE ALLER KANTEN, DIE   *)
                                        (* VON DIESEM KNOTEN WEGFÜHREN    *)
                    NEXT: TKNOTENREF;   (* VERKETTET ALLE KNOTEN IM GRAPH *)
                                        (* ZU EINER LISTE                 *)
                  END;
         TKANTE = RECORD
                    MARKIERUNG: TEINGABE;
                    NACH: TKNOTENREF;   (* ZIELKNOTEN DIESER KANTE        *)
                    NEXT: TKANTENREF;   (* ZEIGER AUF DIE NÄCHSTE KANTE,  *)
                                        (* DIE AM SELBEN KNOTEN BEGINNT   *)
                  END;
  VAR  KNOTENLISTE: TKNOTENREF;    (* ANFANG DER LISTE ALLER KNOTEN       *)
       ANFANG     : TKNOTENREF;    (* ERSTER KNOTEN IM GRAPHEN            *)
       W: ARRAY [1..100] OF CHAR; (* WORT FÜR PRÜFUNG, OB ERKANNT         *)
       I: INTEGER;
  PROCEDURE GRAPHEINLESEN;
  (* KOMPLETTEN GRAPHEN IM SPEICHER AUFBAUEN. ANFANG (GLOBAL) ZEIGT AUF  *)
  (* DEN ERSTEN EINGEGEBENEN KNOTEN. FÜR JEDEN KNOTEN WIRD ISTMARKIERT:= *)
  (* FALSE GESETZT.  AKZEPTIERENDE KNOTEN WERDEN MIT NEGATIVEN NUMMERN   *)
  (* GESPEICHERT.                                                        *)
  (* DIE EINGABE VON DER TASTATUR HAT FOLGENDES FORMAT:  PRO ZEILE EINE  *)
  (* KANTE. NACHEINANDER NUMMER DES AUSGANGSKNOTENS, EIN ZEICHEN UNGLEICH*)
  (* LEERZEICHEN ALS MARKIERUNG UND DIE NUMMER DES ZIELKNOTENS.          *)
  (* DIE LETZTE ZEILE WIRD MIT EINER 0 BEENDET                           *)
     VAR ISTERSTER: BOOLEAN;    (* TRUE BEIM EINLESEN DER ERSTEN KANTE   *)
         VON       : INTEGER;   (* NUMMER DES AUSGANGSKNOTENS            *)
         NACH      : INTEGER;   (* NUMMER DES ZIELKNOTENS                *)
         MIT       : TEINGABE;  (* MARKIERUNG DER KANTE                  *)
         V, N      : TKNOTENREF; (* ZEIGER AUF AUSGANGS- UND ZIELKNOTEN  *)
         KANTE     : TKANTENREF; (* ZEIGER AUF NEU EINZUFÜGENDE KANTE    *)
     FUNCTION SUCHEKNOTEN (N: INTEGER): TKNOTENREF;
     (* SUCHE IN DER LISTE ALLER KNOTEN NACH EINEM KNOTEN, DER DIE NUMMER*)
     (* N BESITZT. DAS FUNKTIONSERGEBNIS IST EIN ZEIGER AUF EINEN RECORD *)
     (* VOM TYP TKNOTEN ODER DER WERT NIL, FALLS KEIN SOLCHER KNOTEN GE- *)
     (* FUNDEN WURDE.                                                    *)
       VAR Q: TKNOTENREF;
     BEGIN
       Q:= KNOTENLISTE;           (* ZEIGER AUF DEN ANFANG DER LISTE      *)
       SUCHEKNOTEN:= NIL;         (* FUNKTIONSERG. VORLÄUFIG FESTLEGEN    *)
       WHILE Q<>NIL DO            (* SOLANGE NOCH KNOTEN IN DER LISTE SIND *)
         IF Q^.NUMMER = N THEN    (* KNOTEN GEFUNDEN, ...                 *)
           BEGIN
             SUCHEKNOTEN:= Q;     (* ZEIGER ALS FUNKTIONSERGEBNIS SETZEN  *)
               Q:= NIL            (* SCHLEIFE BEENDEN                     *)
           END
         ELSE
           Q:= Q^.NEXT;           (* SONST WEITER MIT NÄCHSTEM KNOTEN IN  *)
                                  (* DER LISTE                            *)
     END; (* SUCHEKNOTEN *)
     FUNCTION NEUERKNOTEN (N: INTEGER): TKNOTENREF;
     (* LEGE NEUEN (UNMARKIERTEN) KNOTEN MIT DER NUMMER N AN. DER RECORD *)
     (* WIRD IN DIE LISTE ALLER RECORDS DES GRAPHEN EINGEFÜGT, DAMIT ER  *)
     (* SPÄTER MIT DER FUNKTION SUCHEKNOTEN GEFUNDEN WIRD.               *)
       VAR Q: TKNOTENREF;
     BEGIN
       NEW(Q);                    (* SPEICHERPLATZ FÜR NEUEN RECORD HOLEN *)
       WITH Q^ DO                 (* RECORD KORREKT VORBELEGEN:           *)
         BEGIN
           NUMMER:= N;            (* NUMMER DES KNOTENS                   *)
           ISTMARKIERT:= FALSE;   (* NOCH UNMARKIERT                      *)
           KANTENLISTE:= NIL;     (* BIS JETZT BEGINNT NOCH KEINE KANTE AN *)
                                  (* DIESEM KNOTEN.                       *)
           NEXT:= KNOTENLISTE;    (* Q WIRD AM ANFANG DER KNOTENLISTE EIN- *)
         END;                     (* GEFÜGT.                              *)
       KNOTENLISTE:=Q;            (* Q IST JETZT ERSTER KNOTEN IN KNOTENL. *)
       NEUERKNOTEN:=Q;            (* FUNKTIONSERGEBNIS IST ZEIGER AUF Q^  *)
     END; (* NEUERKNOTEN *)
  BEGIN (* GRAPHEINLESEN *)
    KNOTENLISTE:= NIL;            (* NOCH IST DIE KNOTENLISTE LEER        *)
    ISTERSTER:= TRUE;             (* DIE FOLGENDE KANTE IST DIE ERSTE     *)
    READ(VON);
    WHILE VON<>0 DO               (* ENDE DER EINGABE WIRD DURCH KNOTEN MIT*)
      BEGIN                       (* DER NUMMER 0 GEKENNZEICHNET          *)
        REPEAT
          READ(MIT)               (* MARKIERUNG UNGLEICH LEERZEICHEN LESEN *)
        UNTIL MIT<>' ';
        READLN (NACH);            (* JETZT NOCH DIE NUMMER DES ENDKNOTENS *)
        V:= SUCHEKNOTEN(VON);     (* HOLE ZEIGER AUF DEN AUSGANGSKNOTEN   *)
        IF V=NIL THEN             (* AUSGANGSKNOTEN IST NEU, DESHALB      *)
          V:=NEUERKNOTEN(VON);    (* NEUEN KNOTEN ANLEGEN                 *)
```




```
        N:= SUCHEKNOTEN(NACH);   (* HOLE ZEIGER AUF DEN ZIELKNOTEN          *)
        IF N=NIL THEN            (* FALLS NICHT BEREITS GESPEICHERT,        *)
          N:= NEUERKNOTEN(NACH); (* NEUEN KNOTEN ANLEGEN.                   *)
        NEW(KANTE);              (* BILDE JETZT EINE NEUE KANTE             *)
        KANTE^.MARKIERUNG:= MIT; (* EIN ZEICHEN ALS MARKIERUNG EINTRAGEN    *)
        KANTE^.NACH:= N;         (* KANTE ENTHÄLT ZEIGER AUF ZIELKNOTEN     *)
        (* JETZT KANTE IN DER LISTE DER KANTEN VON KNOTEN V^ EINFÜGEN:      *)
        KANTE^.NEXT:= V^.KANTENLISTE;
        V^.KANTENLISTE:= KANTE;
        IF ISTERSTER THEN        (* BEIM ERSTEN KNOTEN ANFANG SETZEN        *)
          BEGIN
            ISTERSTER:= FALSE;
            ANFANG:= V
          END;
        READ(VON)
     END; (* WHILE *)
   END; (* GRAPHEINLESEN *)
   FUNCTION ISTLEER (Q: TKNOTENREF): BOOLEAN;
   (* DIESE FUNKTION PRÜFT FÜR DEN KNOTEN Q^, OB KEIN AKZEPTIERENDER      *)
   (* KNOTEN ERREICHT WERDEN KANN. IN DIESEM FALL IST ISTLEER FÜR ALLE   *)
   (* VON Q ERREICHBAREN (NICHT MARKIERTEN) KNOTEN EBENFALLS TRUE        *)
   (* AM ANFANG MUSS DAS FELD ISTMARKIERT IM RECORD TKNOTEN FÜR ALLE     *)
   (* KNOTEN AUF FALSE GESETZT WERDEN.                                   *)
     VAR LEER : BOOLEAN;
         NACHF: TKANTENREF;
    BEGIN
      Q^.ISTMARKIERT:= TRUE;    (* MARKIERE KNOTEN, DAMIT DIESER NICHT   *)
                                (* DOPPELT GEPRÜFT WIRD                  *)
      LEER:= Q^.NUMMER>=0;      (* FALLS Q SELBST EIN AKZEPTIERENDER     *)
                                (* KNOTEN IST, KANN NATÜRLICH EIN AKZ.   *)
                                (* KNOTEN ERREICHT WERDEN.               *)
      NACHF:= Q^.KANTENLISTE;   (* ZEIGER AUF DIE ERSTE KANTE, DIE BEI   *)
                                (* KNOTEN Q BEGINNT                      *)
      (* SOLANGE KEIN AKZEPTIERENDER KNOTEN ERREICHT WURDE, VERFOLGE ALLE*)
      (* KANTEN, DIE VON Q WEGFÜHREN:                                    *)
      WHILE LEER AND (NACHF<>NIL) DO
        BEGIN
          IF NOT NACHF^.NACH^.ISTMARKIERT THEN
                                (* FALLS ZIELKNOTEN DER KANTE UNMARKIERT *)
            LEER:= ISTLEER(NACHF^.NACH); (* PRÜFE DISEN ZIELKNOTEN       *)
          NACHF:= NACHF^.NEXT;  (* WEITER MIT DER NÄCHSTEN KANTE         *)
        END;
      ISTLEER:= LEER;           (* FALLS LEER IMMER NOCH TRUE IST, WIRD  *)
                                (* ALSO KEIN AKZEPPT. KNOTEN ERREICHT    *)
   END; (* ISTLEER *)
   FUNCTION ERKANNT (Q: TKNOTENREF; I:INTEGER): BOOLEAN;
   (* PRÜFE, OB VON Q AUS MIT DER BUCHSTABENFOLGE AB W[I] EIN AKZEP-     *)
   (* TIERENDER KNOTEN ERREICHT WIRD.                                    *)
      VAR OK    : BOOLEAN;
          NACHF : TKANTENREF;
    BEGIN
      IF W[I] = ENDE THEN        (* ENDE DES WORTES W ERREICHT:           *)
        ERKANNT:= Q^.NUMMER<0    (* ERKANNT, FALLS Q^ EIN AKZEPTIERENDER  *)
                                 (* ZUSTAND IST.                          *)
      ELSE
        BEGIN
          OK:= FALSE;              (* NOCH IST KEIN AKZ. ZUSTAND GEFUNDEN *)
          NACHF:= Q^.KANTENLISTE;  (* ZEIGER AUF DEN ANFANG DER LISTE DER *)
                                   (* KANTEN, DIE BEI Q BEGINNEN          *)
          (* PROBIERE ALLE KANTEN, DIE BEI Q BEGINNEN UND MIT W[I]        *)
          (* MARKIERT SIND, BIS ERFOLG (OK=TRUE)                          *)
          WHILE NOT OK AND (NACHF<>NIL) DO
            BEGIN
              IF NACHF^.MARKIERUNG=W[I] THEN
                OK:= ERKANNT(NACHF^.NACH, I+1);
                                   (* WEITER MIT DER NÄCHSTE BUCHSTABEN   *)
              NACHF:= NACHF^.NEXT; (* NÄCHSTE KANTE IN DER LISTE          *)
            END;
          ERKANNT:= OK             (* FUNKTIONSERGEBNIS FESTLEGEN         *)
        END; (* IF *)
   END; (* ERKANNT *)
BEGIN (* HAUPTPROGRAMM *)
  GRAPHEINLESEN;                   (* HOLE BESCHREIBUNG DES GRAPHEN       *)
  WRITE('DIE AKZEPTIERTE SPRACHE IST ');
  IF NOT ISTLEER(ANFANG) THEN      (* PRÜFE, OB VOM ANFANGSKNOTEN EIN     *)
                                   (* AKZ. KNOTEN ERREICHBAR IST.         *)
     WRITE ('NICHT ');
  WRITELN ('LEER.');
  REPEAT
    WRITELN ('GEBEN SIE JETZT DAS ZU TESTENDE WORT EIN ($ AM WORTENDE)');
    WRITELN ('(PROGRAMMENDE MIT DEM WORT $)');
    I:= 0;
    REPEAT                         (* STRING W EINLESEN                   *)
      I:= I+1; READ (W[I]);
    UNTIL W[I]=ENDE;
    WRITELN; WRITE ('DAS WORT IST ');
    IF NOT ERKANNT(ANFANG,1) THEN  (* PRÜFE, OB MIT DEM WORT W VOM        *)
                                   (* ANFANGSKNOTEN EIN AKZ. KNOTEN       *)
                                   (* ERREICHBAR IST.                     *)
      WRITE('NICHT ');
    WRITELN('IN DER SPRACHE ENTHALTEN.');
  UNTIL W[1]='$';
END.
```

Listing 3. Das Programm zum Untersuchen von Graphen. Das Zeichen »^« entspricht dem Hochpfeil (»↑«).

ausgehen können. In diesem Fall prüft ERKANNT alle Wege, bis eine Kante zu einem Erfolg führt.

Ähnlich sieht die Lösung der Teilaufgabe 2 aus. Um festzustellen, ob überhaupt ein Wort erkannt wird, genügt es, eine beliebig markierte Kantenfolge vom Anfangsknoten zu einem akzeptierenden Knoten zu finden. Daher wird beim Aufruf ISTLEER für den Knoten Q, der als Parameter übergeben wird, zunächst geprüft, ob Q selbst ein akzeptierender Knoten ist. Wenn ja, so existiert ein akzeptiertes Wort. Ansonsten werden alle Kanten verfolgt, und geprüft, ob man über eine dieser Kanten einen Endzustand erreichen kann. Wiederum ruft sich als ISTLEER rekursiv auf. Jedoch würde ohne weitere Vorkehrung dieser Algorithmus beim Graphen aus Bild 10 in eine Endlosschleife geraten. Vom Ausgangsknoten 1 erreicht man den Knoten 2. Da man alle Kanten von 2 verfolgt, gelangt man über die Kante »B« zurück zu Knoten 1. Von dort springt man wieder zu Knoten 2 und so weiter. Die Lösung besteht darin, daß man alle bereits besuchten Knoten markiert. Dazu wird der Record um das Feld

```
ISTMARKERT: BOOLEAN
```

erweitert. Bei der Eingabe des Graphen sind alle Knoten unmarkiert. Erreicht man jedoch in der Funktion ISTLEER den Knoten Q, so wird dieser mit

```
Q↑ :ISTMARKERT:= TRUE
```

gekennzeichnet. Bevor man nun einen rekursiven Aufruf der Funktion ISTLEER ausführt, wird zunächst wieder geprüft, ob der Knoten nicht bereits untersucht und markiert wurde.

Das größte Problem ist sicherlich die Eingabe eines Graphen. In diesem Beispiel wird eine kantenorientierte Eingabe vorgenommen: Der Graph in Bild 8 wird folgendermaßen beschrieben:

```
 1 A  2
 2 B -3
-3 A  2
0
```

Jede Zeile benennt also die Nummer des Anfangsknoten, die Markierung der Kante und die Nummer des Endknotens der Kante. Das Ende der Eingabe markiert die Zahl Null. Die Eingabe nimmt die Prozedur EINLESEN vor. Zunächst wird die Nummer des Ausgangsknotens (VON) gelesen, anschließend alle Leerzeichen ignoriert, bis die Markierung (MIT) indentifiziert wurde. Daran schließt sich die Nummer des Endknotens an. Akzeptierende Knoten sind wieder durch negative Zahlen kenntlich.

Für jeden Knoten wird zunächst festgestellt, ob bereits ein Record vom Typ TKNOTEN mit dieser Nummer existiert. Die Funktion SUCHEKNOTEN liefert zu einer Nummer N einen Zeiger auf einen

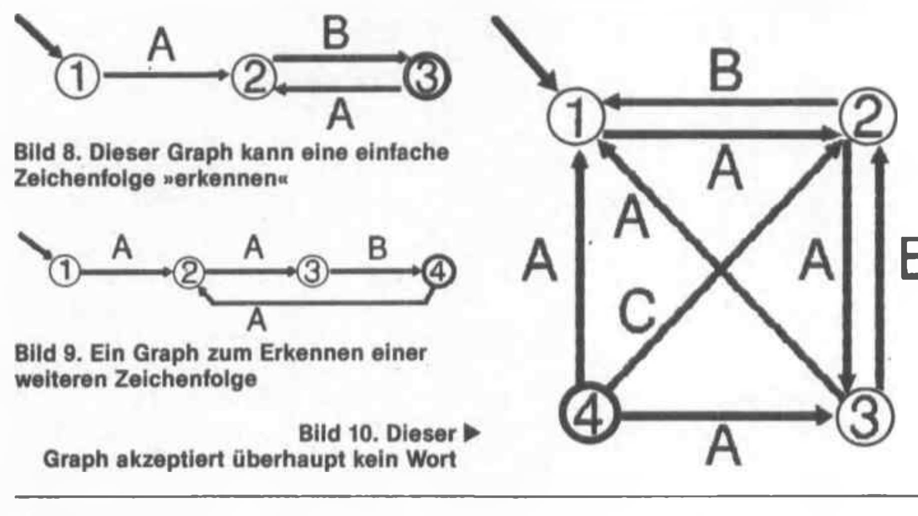


Bild 8. Dieser Graph kann eine einfache Zeichenfolge »erkennen«

Bild 9. Ein Graph zum Erkennen einer weiteren Zeichenfolge

Bild 10. Dieser ▶ Graph akzeptiert überhaupt kein Wort

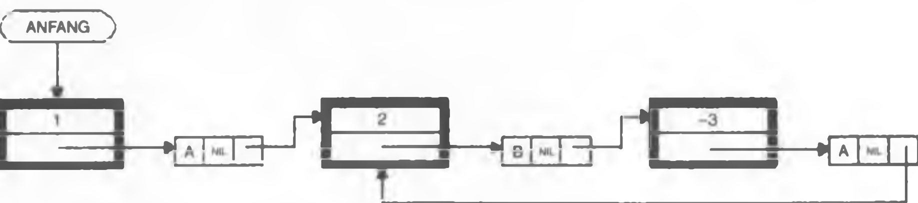


Bild 11. Interne Zeigerdarstellung des Graphen aus Bild 8

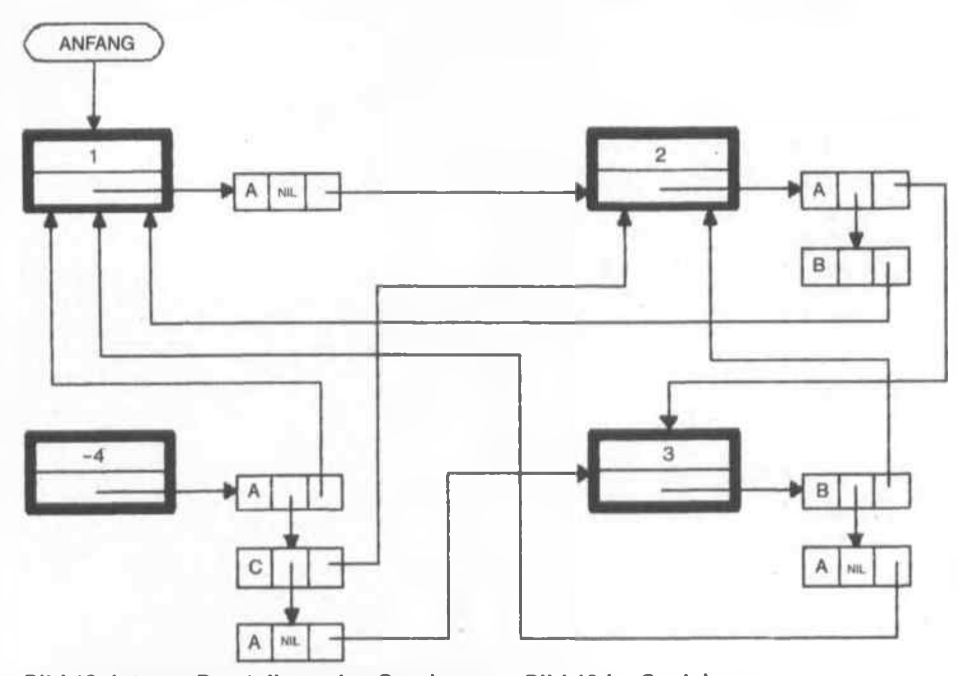


Bild 12. Interne Darstellung des Graphen aus Bild 10 im Speicher

```
 1 A 2
 2 B -3
-3 A 2
 0
DIE AKZEPTIERTE SPRACHE IST NICHT LEER.
ABAB$
DAS WORT IST IN DER SPRACHE ENTHALTEN
ABA$
DAS WORT IST NICHT IN DER SPRACHE ENTHALTEN
AA$
DAS WORT IST NICHT IN DER SPRACHE ENTHALTEN.
$
```

```
 1 A 2
 2 A 3
 3 B -4
-4 A 2
0
DIE AKZEPTIERTE SPRACHE SIT NICHT LEER
AAB$
DAS WORT IST IN DER SPRACHE ENTHALTEN.
AABAAB$
DAS WORT IST IN DER SPRACHE ENTHALTEN
A$
DAS WORT IST NICHT IN DER SPRACHE ENTHALTEN.
$
```

```
 1 A 2
 2 B 1
 2 A 3
 3 B 2
 3 A 1
-4 A 1
-4 C 2
-4 A 3
0
DIE AKZEPTIERTE SPRACHE IST LEER
$
```

Bild 13. Einige Probeläufe für den Graphen aus Bild 8

Knoten-Record, oder den Wert NIL, falls noch kein solcher Record vorkam. Eventuell muß also mit der Funktion NEUERKNOTEN ein Knoten mit der Nummer N angelegt werden. Die Funktion liefert einen Zeiger auf den neuen Record, der außerdem als unmarkiert gekennzeichnet wird.

Anschließend kann in EINLESEN ein neuer Record vom Typ TKANTE mit der angegebenen Markierung angelegt werden. Dabei wird im Feld NACH ein Zeiger auf den Endknoten N (mit der Nummer NACH) eingetragen und schließlich dieser Kanten-Record in die Liste der Kanten des Knotens V (mit der Nummer VON) eingefügt.

Da der Ausgangsknoten immer als erster in der Eingabe genannt wird, benutzt man die boolesche Variable ISTERSTER. Falls ISTERSTER=TRUE ist, muß also der Zeiger ANFANGSKNOTEN noch auf den Record V1 gesetzt werden.

Einige Probeläufe des Programmes mit verschiedenen Graphen zeigt Bild 13.

Natürlich ist nicht zu erwarten, daß ein absoluter Anfänger den gesamten Artikel sofort in die Praxis umsetzen kann. Die letzten Beispiele zeigen jedoch deutlich, daß Pascal für kompliziertere Aufgaben einen deutlichen Leistungsvorsprung gegenüber anderen Sprachen besitzt, die entweder nur einfache Datenstrukturen aufweisen (Basic, Fortran, Cobol), oder aber keine so wirkungsvolle Abstraktion von der internen Darstellung der Daten erlauben, so daß die Typüberprüfung in Ausdrücken und bei der Parameterübergabe auf den Programmierer abgewälzt wird (C und Forth).

(Florian Matthes/ev)

Info: Empfehlenswerte Pascal-Literatur:

F Matthes, »Pascal mit dem C 64«, Markt&Technik Verlag (mit Pascal-Compiler aus Diskette)

N. Wirth, »Algorithmen und Datenstrukturen«, Teubner Studienbücher

K Mehlhorn, »Effiziente Algorithmen«, Teubner Studienbücher

K. Jensen, N. Wirth, »Pascal, User Manual and Report, Lecture Notes in Computer Science, Vol. 18«, Springer Verlag (Berlin)

H. Schauer »Pascal für Anfänger«, R. Oldenbourg Verlag

E. Kaucher, R. Klatte, Ch. Ullrich, »Programmiersprachen im Griff, Band 2: Pascal 4, BI-Hochschultaschenbücher

R. Busch, »Der sichere Einstieg in Pascal«, Franzis' Verlag

R.-D. Klein, »Was ist Pascal?«, Franzis' Verlag

J.N.P. Hume, R.C. Holt, »UCSD-Pascal«, Markt&Technik Verlag

D.T. Barnard, Crawford, »Pascal – Probleme und Anwendungen«, Markt&Technik Verlag

I. Lüke, P. Lüke, »Turbo Pascal«, Markt&Technik Verlag

»Turbo Pascal Handbuch«, Heimsoeth Software (mit Turbo Pascal Compiler auf Diskette)

»Turbo Tutor«, Heimsoeth Software (mit Demo-Diskette)

K.H. Rollke, »Das Turbo Pascal Buch«, Sybex-Verlag

A. Luehrmann, H. Pechham, »Appel II Pascal«, Tewi-Verlag

Prof. Nestle, E. Ostertag, »Kleiner Sprachführer Basic-Logo-Pascal«, Markt&Technik Verlag

K. Mikitta, »Pascal für Schulen«, Aulis-Verlag

I.R. Wilson, A.M. Addyman, »Pascal«, Carl-Hauser-Verlag

K.H Rollke, »Grundkurs in Pascal, Band 1 und 2«, Sybex-Verlag

R. Zaks, »Einführung in Pascal und UCSD-Pascal«, Sybex-Verlag

J. Tiberghien, »Das Pascal Handbuch«, Sybex-Verlag

Cursor-Kur mit Turbo-Pascal

Der unstet flackernde Cursor des C 128 ist vielen CP/M-Anwendern ein Dorn im Auge. Hier naht Abhilfe in Form eines Turbo-Pascal-Programms.

Commodore hat beim neuen Basic 7.0 an den gestreßten Programmierer gedacht und gleich mehrere mögliche Cursor-Arten vorprogrammiert, die auf einfache Weise, nämlich durch Drücken der Escape-Taste und eines Buchstabens, für den Anwender erreichbar sind. So ist sowohl ein blinkender Strich- oder Unterstreichungs-Cursor wie auch ein stehender Block-Cursor machbar (siehe Tabelle).

Anders allerdings sieht es im CP/M-Modus des Computers aus. Hier sind dem Programmierer die Hände gebunden. Er muß sich wohl oder übel den närrisch flackernden Cursor gefallen lassen.

Unter CP/M sind Eingriffe in das Betriebssystem des Computers weitaus schwieriger zu realisieren als unter Basic. Für die Mehrheit der Commodore-Besitzer ist die Maschinensprache des Z80-Prozessors wohl mehr als ein Buch mit sieben Siegeln, so daß ein direkter Zugriff per Assembler ausscheidet. Außerdem erfordern Eingriffe in das Betriebssystem unter CP/M genaueste Kenntnisse des CP/M-Systems. Auch diese Kenntnisse dürften bei vielen Anwendern fehlen.

Aber dies ist trotzdem noch kein Grund zum Verzweifeln, denn wozu gibt es höhere Programmiersprachen? Aus obengenannter Not entstand das folgende Programm in Turbo-Pascal 3.0. Es bietet dem Benutzer die Möglichkeit, sich seinen Cursor nach eigenen Wünschen menügesteuert anzupassen. Das Programm kann nach dem Eingeben in ein COM-File compiliert werden, so daß man es ohne Probleme auf all seinen Disketten installieren kann. Es besteht sogar die Möglichkeit, das Programm von PROFILE.SUB aus aufzurufen, damit spart man sich das Laden nach dem BOOTen. Nach der Meldung »Cursor-Mode-Swap für C 128 und CP/M 3.0« kann man sich durch zwei oder drei Tastendrucke seinen persönlichen Cursor installieren.

Noch etwas zum Abtippen: Das Listing wurde im DIN-Modus der Tastatur eingegeben. Daher konnte zwar mit deutschen Umlauten gearbeitet werden, die geschweiften Kommentarklammern und die eckigen Mengenklammern wurden aber durch die von Turbo-Pascal ebenfalls zugelassenen Ersatzzeichen »(*« und »*)« (für geschweifte Klammern) beziehungsweise »(.« und ».)« (für eckige Klammern) ersetzt.

(Udo Reetz/ev)

Cursor-Mode	Escape-Sequenz
Block	ESC + S
Strich	ESC + U
Blinken	ESC + F
Stehen	ESC + E

Tabelle der möglichen Cursor-Formen im Basic 7.0

```
(*$u+*)

(**********************************************************)
(*   Beispiel für Zugriff auf die I/O-Ports des 8510:    *)
(* Utility zum komfortablen ändern des VDC Cursor Modes *)
(*       Turbo Pascal auf Commodore 128 unter CP/M       *)
(**********************************************************)

program cursorset (input, output);

var BlinkMaske,          (* Maske für Blinkbits in Reg. 10 *)
    StartLine: byte;     (* Rasterstartzeile des Cursors *)
           c: char;

procedure PortOut (adr: integer; wert: byte);
   (* Ausgabe von wert auf dem 16-Bit-Port adr *)
begin
  inline ($ed/$4b/adr/       (* LD   BC,(adr)   *)
          $3a/wert/          (* LD   A,(wert)   *)
          $ed/$79            (* OUT  (C),A)     *)
         )
end (* procedure PortOut *);

procedure PortIn (adr: integer; var wert: byte);
   (* Lesen von Port adr nach wert *)
begin
  inline ($ed/$4b/adr/       (* LD   BC,(adr)   *)
          $ed/$78/           (* IN   A,(C)      *)
          $dd/$2a/wert/      (* LD   IX,(wert)  *)
          $dd/$77/$00/       (* LD   (IX+0),A   *)
          $dd/$36/$01/$00    (* LD   (IX+1),0   *)
         )
end (* procedure PortIn *);

function vdcIn (reg: byte): byte;
   (* Liefert den Wert des VDC-Registers reg *)
var x: byte;    (* Hilfsvariable *)
begin
  PortOut ($d600, reg);      (* Registeradresse an VDC *)
  repeat
    PortIn ($d600, x)        (* Status lesen *)
  until x>=128;
  PortIn ($d601, x);         (* Daten-Register lesen *)
  vdcIn := x
end (* function vdcIn *);

procedure vdcOut (reg, wert: byte);
   (* Schreibt wert in VDC-Register reg *)
var x: byte;    (* Hilfsvariable *)
begin
 PortOut ($d600, reg);      (* Registeradresse an VDC *)
 repeat
    PortIn ($d600, x)        (* Status lesen *)
  until x>=128;
  PortOut ($d601, wert)      (* Daten-Register schreiben *)
end;

begin (* main *)
  clrscr;
  writeln ('Cursor-Mode-Swap für C 128 und CP/M 3.0': 20);
  writeln ('=======================================': 20);
  writeln; writeln; writeln; writeln;
  write ('      Cursor B)linkend oder S)tehend ?');
  repeat
    read (kbd, c);
  until c in (. 'b', 'B', 's', 'S' .);
  writeln (c); writeln;
  if c in (. 's', 'S' .)
    then BlinkMaske := 0
    else
      begin
        write ('      S)chnell oder N)ormal blinken ? ');
        repeat
          read (kbd, c);
        until c in (. 's', 'S', 'n', 'N' .);
        writeln (c); writeln;
        if c in (. 's', 'S' .)
          then BlinkMaske := $40
          else BlinkMaske := $60
      end (* else *);
  write ('      U)nderline- oder B)lockcursor ? ');
  repeat
    read (kbd, c);
  until c in (. 'u', 'U', 'b', 'B' .);
  writeln (c); writeln;
  if c in (. 'u', 'U' .)
    then StartLine := 7
    else StartLine := 0;
  writeln; writeln;
  writeln ('**** Änderungen durchführen ? ****');
  repeat
    read (kbd, c)
  until c in (. 'j', 'J', 'n', 'N' .);
  writeln (c); writeln;
  if c in (. 'j', 'J' .)
    then
      begin
        vdcOut (10, BlinkMaske or StartLine);
        vdcOut (11,7)
      end (* else *)
end. (* program cursorset *)
```

Listing zu unserer Cursor-Kur

Kaiser

Möchten Sie auch einmal wie ein König ein Land regieren und das Staatsgeschick lenken? Dann versuchen Sie, als Kaiser Ihr Reich instand zu halten und für Wohlstand zu sorgen.

Kaiser oder auch Hammurabi ist eine Strategie-Simulation. Ihre Aufgabe ist es, durch Landkäufe oder -verkäufe, Verkauf von Getreide optimale Ausnutzung der Anbauflächen und gerechte Verteilung der Nahrungsmittel an die Bevölkerung das Reich in bestem Zustand zu erhalten. So kann zum Beispiel das Land zugrunde gehen, wenn die Bevölkerung nichts zu essen hat. Andererseits wollen die Leute eben mit vollem Bauch weniger arbeiten, so daß die Produktivität absinkt. Regeln Sie alles so, daß es Ihrem Reich wohler geht, so daß Sie lange regieren können.

Das Programm

Das Programm »Kaiser« (Listing) ist nur ein kleines, aber lauffähiges Grundgerüst, das ohne weiteres ausbaufähig ist. So können noch Faktoren wie arbeitsfaule Bevölkerung, Überfälle brandschatzender Horden, Unwetterkatastrophen oder vieles mehr mit integriert werden, um das Spiel reizvoller zu gestalten. Je komplexer die Faktoren sind, desto attraktiver wird auch das Spiel selbst.

Die Random-Funktion, mit der hier gearbeitet wird, ist nicht in jedem Pascal implementiert. Fehlt sie bei Ihnen, so müssen Sie noch eine Prozedur »random« mit in das Programm einbauen. Spielen Sie ruhig etwas mit dem Programm und bauen Sie von Zeit zu Zeit noch weitere Faktoren ein. Sie werden sehen: Durch viele Ereignisse, die das Geschehen beeinflussen können, gewinnt Kaiser immer mehr an Reiz.

(Dieter Mayer/hi)

```
PROGRAM kaiser;

CONST
   produktivitaet  = 7;

VAR
   anbauflaeche,gesamtflaeche,landkauf,
   vorrat,flaecheprokopf,rattenschaden,
   hektarertrag,ernte,zuteilung,nahrungprokopf,
   geburtenrate,sterberate,ausgaben,kornpreis,
   verkauf,vermoegen:real;
   jahr,jahre:0..20;
   zaehler,volk,zahl,geborene,
   gestorbene,bodenpreis:integer;
   getreide,kaufen,sinnvoll:boolean;
   input:char;

PROCEDURE initialisierung;
BEGIN
   zaehler         :=1;
   kornpreis       :=3;
   gesamtflaeche   :=30000;
   anbauflaeche    :=3000;
   bodenpreis      :=30;
   vermoegen       :=20000;
   hektarertrag    :=4;
   vorrat          :=20000;
   ernte           :=12000;
   volk            :=4000;
   flaecheprokopf:=gesamtflaeche/volk;
   geborene        :=0;
   gestorbene      :=0;
   nahrungprokopf:=20;
   ausgaben        :=vermoegen/8;
END;

PROCEDURE berechnung;
BEGIN
    IF kaufen = true THEN
     BEGIN
      gesamtflaeche:=gesamtflaeche+landkauf;
      vermoegen:=vermoegen-(landkauf*bodenpreis);
     END
                  ELSE
     BEGIN
      gesamtflaeche:=gesamtflaeche-landkauf;
      vermoegen:=vermoegen+(landkauf*bodenpreis);
     END;
bodenpreis:=28+random(5);
IF getreide = true THEN
 BEGIN
  vorrat:=vorrat-verkauf;
  vermoegen:=vermoegen+(kornpreis*verkauf)
 END;
vorrat:=vorrat-(zuteilung*volk);
   ausgaben:=random(6000);
   vermoegen:=vermoegen-ausgaben;
   kornpreis:=1+random(3);
   hektarertrag:=4+random(3);
   ernte:=anbauflaeche*hektarertrag;
   vorrat:=vorrat+ernte;
   rattenschaden:=abs(random(50)/100);
   vorrat:=abs(vorrat*1-rattenschaden);
   flaecheprokopf:=gesamtflaeche/volk;
   geborene:=random(90);
   gestorbene:=random(25);
   volk:=volk+geborene-gestorbene;
END;

PROCEDURE zustandsausgabe;
   BEGIN
      writeln('Momentaner Spielstand:');
      writeln;
      write('Wir schreiben das ');
      IF zaehler>jahre THEN zaehler:=jahre;
      write(zaehler);
      writeln('. Jahr Ihrer Herrschaft.');
      writeln;
      write('Ihr Volk besteht zur Zeit aus ');
      write(volk);
      writeln(' Einwohnern, wobei Sie');
      write(geborene);
      write(' Geburten und ');
      write(gestorbene);
      writeln(' verstorbene Personen haben.');
      write('Sie besitzen ');
      write(gesamtflaeche:8:2);
      write(' Hektar Land, wovon ');
      write(anbauflaeche:8:2);
      writeln(' Hektar bebaut sind.');
      write('Jeder Buerger bewohnt durchschnittlich ');
      write(flaecheprokopf:6:2);
      writeln(' Hektar Boden.');
      write('Der Vorrat belaeuft sich auf ');
      write(vorrat:10:2);
      writeln(' DZ Getreide.');
      write('Die Staatskasse enthaelt ');
      write(vermoegen:10:2);
      writeln(' Taler.');
      write('Staatsausgaben dieses Jahr: ');
      write(ausgaben:9:2);
      writeln(' Taler.');
      write('Heutiger Bodenpreis: ');
      write(bodenpreis);
      writeln(' Taler je Hektar.');
      write('Jahresertrag:  ');
      write(hektarertrag:8:2);
      writeln(' DZ Getreide je Hektar.');
      write('Kornpreis    :     ');
      write(kornpreis:5:2);
      writeln(' Taler je DZ.');
      write('Gesamternte :  ');
      write(ernte:8:2);
      writeln(' DZ Getreide.');
      write('Durch Ratten wurden ');
      write(rattenschaden:5:2);
      writeln(' % der Ernte vernichtet.');
     zaehler:=zaehler+1;
END;

PROCEDURE eingabe;
   BEGIN
      writeln;
      write('Moechten Sie Land kaufen ');
      write('oder verkaufen (K/V) ? ');
      read(input);
      writeln;
      IF input ='k' THEN kaufen:=true
                 ELSE kaufen:=false;
      write('Wieviel Hektar ? ');
      read(landkauf);
      writeln;
      write('Gedenken Sie, Getreide ');
      write('zu exportieren (J/N) ? ');
      read(input);
      writeln;
      IF input ='j' THEN
                      BEGIN
```

```
                              getreide:=true;
                              write('Wieviel DZ ? ');
                              read(verkauf);
                              writeln;
                           END
                         ELSE getreide:=false;
      write('Wieviele DZ Getreide goennen ');
      write('Sie jedem Untertan ? ');
      read(zuteilung);
      writeln;
      write('Wieviele Hektar duerfen ');
      write('die Bauern bewirtschaften ? ');
      read(anbauflaeche);
      writeln;
   END;

PROCEDURE eingabepruefung;
BEGIN
   writeln;
   IF kaufen = true THEN
     IF (landkauf*bodenpreis) >= vermoegen THEN
      BEGIN
         write('Landkauf erlaubt Ihre ');
         writeln('Staatskasse nicht.');
         sinnvoll:=false;
      END;
   IF kaufen = false THEN
   IF gesamtflaeche-landkauf <0 THEN
    BEGIN
       writeln('Soviel Land besitzen Sie nicht.');
       sinnvoll:=false;
    END;
 IF zuteilung <=0 THEN
    BEGIN
       write('Leuteschinder - Sollen ');
       writeln('die Buerger verhungern ?');
       sinnvoll:=false;
    END;
 IF zuteilung >= (vorrat/volk) THEN
    BEGIN
       writeln('So gross sind Ihre Vorraete nicht.');
       sinnvoll:=false;
    END;
   IF anbauflaeche <=0 THEN
      BEGIN
         write('Spassvogel - wo wollen ');
         writeln('Sie denn anbauen ?');
         sinnvoll:=false;
      END;
   IF anbauflaeche >0.5*(gesamtflaeche+landkauf) THEN
      BEGIN
         write('Nur die Haelfte des Landes ');
         writeln('kann bebaut werden.');
         sinnvoll:=false;
      END;
   IF anbauflaeche>produktivitaet*volk THEN
      BEGIN
         write('Soviel koennen Ihre Leute ');
         writeln('nicht bebauen.');
         sinnvoll:=false;
      END;
END;
BEGIN
   initialisierung;
   writeln('Wieviele Jahre moechten Sie regieren ?');
   readln(jahre);
   zahl:=random(99)+1;
   FOR jahr:=1 TO jahre DO
      BEGIN
         zustandsausgabe;
         REPEAT
            BEGIN
            sinnvoll:=true;
            eingabe;
            eingabepruefung;
            END;
         UNTIL sinnvoll;
         berechnung;
      END;
   zustandsausgabe;
END.
```

Listing »Kaiser«: Bestimmen Sie über das Wohl und Wehe Ihres Volkes

Berechnen gemeinsamer Teiler

Dieses Programm nimmt Ihnen die überaus lästige Arbeit ab, alle Zahlen herauszufinden, durch die eine andere teilbar ist.

Sicher hatten Sie schon gelegentlich mal das Problem, daß Sie alle Teiler einer Zahl herausfinden wollten. Sei es für die Verwendung im Bruchrechnen oder zur Ermittlung von Primzahlen. Diese Arbeit nimmt Ihnen nun die Prozedur »Teiler« (Listing) ab.

Nach der Compilation werden Sie gefragt, aus welcher Zahl alle Teiler berechnet werden sollen. Bitte geben Sie hier nur ganze Zahlen ein. Das Programm gibt Ihnen nun alle Teiler dieser Zahl aus. Wird kein Teiler gefunden, so ist es eine Primzahl, was auch ein Antwortsatz bekannt gibt.

Zum Programm

Nach der Eingabe der Zahl, aus der die Teiler berechnet werden sollen, wird zuerst ein Flag (Schalter) auf »unwahr« gesetzt. Danach versucht das Programm, die eingegebene Zahl so oft wie möglich zu dividieren. Sind Teiler vorhanden, werden diese ausgegeben und das Flag auf »wahr« gesetzt. Am Ende der Prozedur wird noch einmal der Zustand des Flags abgefragt. Ist es »unwahr«, so konnte kein Teiler gefunden werden, und die Meldung »Diese Zahl ist eine Primzahl« erscheint auf dem Bildschirm.

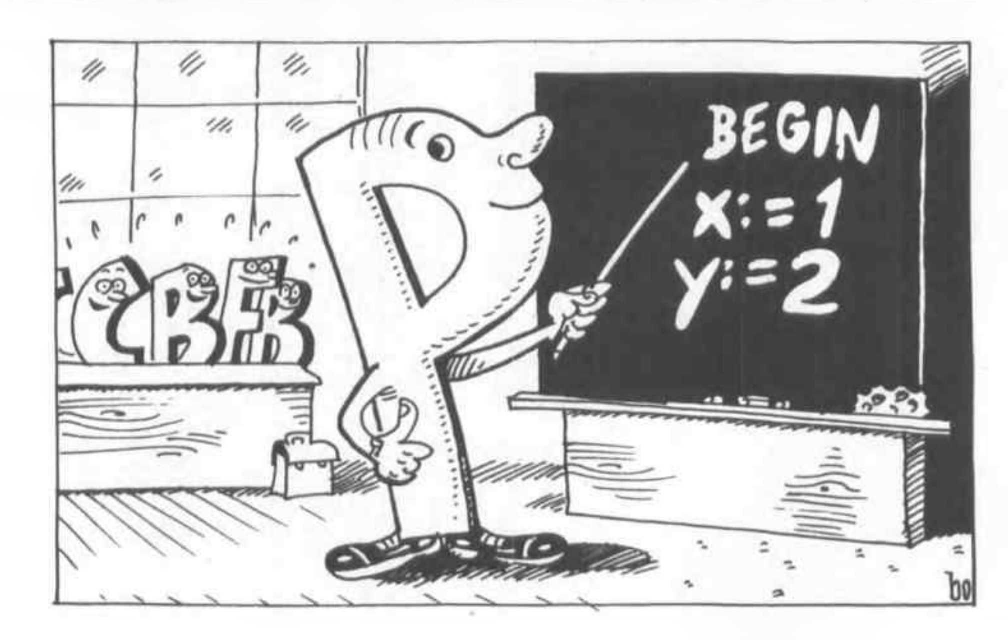
Die Prozedur eignet sich gut zum Einbinden in mathematische Programme, also etwa in ein Mathe-Lexikon auf dem Computer. (Dieter Mayer/hi)

```
PROGRAM Teiler (Input,Output);
VAR Zahl,Teiler:Integer;
    Schalter:boolean;
BEGIN
  write   ('Bitte geben Sie die Zahl ein,
           aus der die Teiler berechnet ');
  writeln ('werden sollen !');
  read (Zahl);
  Schalter:=FALSE;
WHILE Zahl>0 DO
    BEGIN
      FOR Teiler:=2 TO TRUNC(Zahl/2) DO
        BEGIN
          IF Zahl MOD(Teiler)=0
          THEN BEGIN write (Teiler:5);
                     Schalter:=TRUE
               END;
        END;
      IF Schalter=FALSE
       THEN write (' Diese Zahl
                   ist eine Primzahl');
      writeln; writeln;
      read (Zahl);
      Schalter:=FALSE;
    END;
END.
```

Listing »Teiler«

Filer für Turbo-Pascal

Werden lauffähige Turbo-Pascal-Programme erzeugt, so verbraucht die in jedem xxx.com-File vorhandene Run-Time-Library 8 KByte.



Sollen mehrere kurze Programme auf einer Diskette gespeichert werden, stößt man schnell an die Grenze der verfügbaren Speicher-Kapazität. Der Turbo-Pascal-Compiler bietet mit der H-Option die Möglichkeit, Programme ohne die Run-Time-Library zu compilieren und sie dann als sogenannte Chain-Files aufzurufen. Von dieser Möglichkeit, Platz zu sparen, macht das vorliegende Programm Gebrauch.

Die Benutzer-Programme werden speicherplatzsparend als xxx.chn-Files direkt auf die Filer-Diskette compiliert, ins Inhaltsverzeichnis (Inhalt.fil) eingelesen und dann von Filer aus aufgerufen. Um wieder zu Filer zurückkehren zu können, sollten die Benutzer-Programme mit der Prozedur »Ende« (siehe Listing) abgeschlossen werden.

Die Auswahl der einzelnen Funktionen von Filer erfolgt mit den Cursor-Tasten »Pfeil nach oben«, »..nach unten«, »..rechts und links«. Die jeweilige Funktion beziehungsweise die Benutzer-Programme werden invers als Leuchtfeld dargestellt. Durch Drücken der Home-Taste wird die angezeigte Funktion (Benutzer-Programm) ausgeführt.

»Programm« führt die Benutzer-Programme aus, »Neuaufnahme« liest ein auf der Diskette vorliegendes xxx.chn-Programm in das Inhaltsverzeichnis ein, »Löschen, Umbenennen und CP/M« erklären sich von allein und bedürfen somit keiner weiteren Erläuterung.

Ob groß, ob klein...

Alle vom Programm geforderten Eingaben können beliebig als Klein- oder Großbuchstaben erfolgen. Durch Drücken der ESC-Taste können alle laufenden Routinen abgebrochen werden.

Das Quellprogramm besteht aus filer.pas und lib.inc. In lib.inc können Anpassungen an verschiedene Terminals vorgenommen werden. Die Erläuterungen sind in das Programm-Listing eingefügt. Das lauffähige Programm besteht aus filer.com. Das File inhalt.fil wird beim ersten Programm-Lauf automatisch erzeugt.

(M.Schlösser/ev)

```
program filer;

(* Written by M.A. Schlösser        *)
(*            Rheinstr. 20          *)
(*            6100 Darmstadt        *)
(*            Tel: 06151 - 2 25 38  *)
(* Stand vom 25. Juni 1985          *)

(*****************************************************)
(* Turbo-Pascal Programme können speicherplatz-sparend *)
(* mit der H-Option des Compilers compiliert und von   *)
(* diesem Programm aus aufgerufen werden.              *)
(*****************************************************)

type inhaltrec = record
                 Progname: string (.12.);
                 end;

var   dateibuffer: inhaltrec;
      datei      : file of inhaltrec;
type
defString = array (.1..32.) of string (.12.);
defset    = set of char;
anystring = string (.80.);
const
bell       = ^G;  (* Glocke *)

var
lines, inharray        : defstring;
arraynr, arraylaenge,i : integer;
filename,helpstring    : anystring;
pruef,file_exist       : boolean;
filevar                : file:
zch                    : char;

(*$I filer.lib *)

procedure initinhalt (var z       : defstring;
                      var anz_rec: integer);
(* liest Inhaltsverz. von der Diskette und *)
(* erzeugt inharray                        *)
begin
anz_rec:= 0;
pruef:= exist ('a:inhalt.fil');
assign (datei,'a:inhalt.fil');
if pruef then
   begin
   reset(datei);
   while not eof (datei) do
         begin
         read (datei,dateibuffer);
         anz_rec:= anz_rec+1;
         z(.anz_rec.):= dateibuffer.progname;
         end; (* while *)
```

Listing. »filer.pas«

Fortsetzung auf Seite 68

```
      if anz_rec=0 then
         begin
         lowvideo;
      gotoxy(1,12);write ('Neuaufnahme aufrufen '); normvideo;
      end; (* if anz-rec *)
   end  (* if pruef *)
 else begin
      rewrite(datei);
      lowvideo;
      gotoxy(1,12);write ('Neuaufnahme aufrufen '); normvideo;
      end; (* else *)
close (datei);
end; (* procedure initinhalt *)


procedure head_string (var z:defstring);
(* Dient zur Zuordnung der Daten für den *)
(* auszudruckenden array                 *)

begin
z(.1.):= 'Programm'; z(.2.):= 'Neuaufnahme'; z(.3.):='Löschen';
z(.4.):= 'Umbenennen'; z(.5.):= 'CP/M';
end; (* procedure head_string *)


procedure programm_ueberschrift;
begin
gotoxy(1,16);clreol;
write ('PROGRAMME  (vorh: ',arraylaenge,'   max: 32 )');
end;


procedure msg(zeile:integer);
begin
gotoxy(1,zeile);
writeln ('Leuchtfeld mit Cursor-Tasten auf gewünschtes Programm stellen');
writeln ('und die HOME-Taste drücken.');
end;


procedure init_array;
(* Schreibt inharray auf Diskette *)

begin
assign (datei,'inhalt.fil');
rewrite (datei);
for i:= 1 to arraylaenge do
    begin
    dateibuffer.progname := inharray(.i.);
    write(datei,dateibuffer);
    end; (* for i *)
close (datei);
end;  (* procedure init_array *)

(*******************************************)
(*        Hauptprogramm Filer             *)
(*******************************************)
begin
clrscr;
writeln('          Filer für Turbo-Pascal Programme unter CP/M');
writeln('          written by M.A.Schlösser          Juni 1985');
writeln;
for i:= 1 to 79 do write ('-');
initinhalt (inharray,arraylaenge);
head_string (lines); (* liest Funktion-Auswahl in das array lines *)
schreib_array (lines, 5,   1,5,     15,       4           );
            (* array/ anz/ x-y pos/ abstand/  worte/zeile *)
gotoxy(1,7); for i:= 1 to 79 do write ('-');
programm_ueberschrift;
schreib_array (inharray,arraylaenge,1,17,15,4);
(* schreibt Benutzer-Prog. *)
repeat (* until true = false *)
msg(9);
arraynr:= select (lines, 5,   1,5,     15,       4           );
            (* array/ anz/ x-y pos/ abstand/  worte/zeile *)
clrln(9,12);
case arraynr of
 1: begin (* Programm *)
    gotoxy(1,9);lowvideo;
    writeln ('Programm-Auswahl');normvideo;
    msg(10);
    arraynr:= select (inharray,arraylaenge,1,17,15,4);
    if arraynr <> 0 then
       begin
       assign(filevar,inharray(.arraynr.)+'.chn');
       chain (filevar);
       end (* if arraynr *)
     else clrln(9,11);
    end; (* case of 1 *)
 2: begin  (* Neuaufnahme *)
    gotoxy(1,9);Lowvideo;writeln('Neuaufnahme');normvideo;
    write ('Das ins  Inhaltsverzeichnis  aufzunehmende  File muß ');
    writeln ('unter xxx.chn auf der');
    writeln ('Diskette vorliegen. Die Namen-Eingabe ohne File-Kennung!');
    line_editor ('Programm-Namen eingeben',1,13,    8,     filename,
    (.'0'..'9','A'..'z'.));
    (* Text-Ausgabe/ x-y pos/ Länge/ variable/ gültige Zeichen    *)
    if filename <> '' then
       begin
       Umw_in_Grossbuchstaben (filename);
       helpstring:= filename+'.chn';
       pruef:= exist (helpstring);
       if pruef then
          begin
          arraylaenge:=arraylaenge+1;
          inharray(.arraylaenge.):=filename;
          programm_ueberschrift;
          schreib_array (inharray,arraylaenge,1,17,15,4);
          init_array;
          end
       else begin
            write (bell);
            gotoxy(1,13);clreol;
            write ('Das File --> ',helpstring,'
                  <-- ist nicht auf Diskette');
           write (' vorhanden. Taste drücken.');
           repeat until keypressed;
           end; (* else *)
     end; (* if filename <> '' *)
     clrln (9,14);
     end; (* case of 2 *)
 3: begin (* Löschen *)
    gotoxy(1,9);lowvideo; write ('Löschen'); normvideo;
    msg(10);
    arraynr:= select (inharray,arraylaenge,1,17,15,4);
    if arraynr<>0 then
            begin
            clrln(10,11);
            gotoxy(1,10);write ('File -->',inharray(.arraynr.),
                            '<-- löschen (j/n) ');
            repeat
            read (kbd,zch);
            until upcase(zch) IN (.'J','N'.);
            if upcase(zch) = 'J' then
               begin
               assign (filevar,inharray(.arraynr.)+'.chn');
               erase (filevar);
               for i:= arraynr to arraylaenge do
                   inharray(.i.):= inharray (.i+1.);
               arraylaenge:= arraylaenge-1;
               init_array;
               clrln(17,24);
               schreib_array (inharray,arraylaenge,1,17,15,4);
               end; (* if zch = J *)
            end; (* if arraynr <> 0 *)
         clrln(9,11);
         programm_ueberschrift;
         end; (* case löschen *)
     4: begin (* Umbenennen *)
```

Listing. »filer.pas« (Fortsetzung)

```
   gotoxy(1,9);lowvideo; write ('Umbenennen'); normvideo;
   msg (10);
   arraynr:= select (inharray,arraylaenge,1,17,15,4);
   if arraynr <> 0 then
      begin
      file_exist:= false;
      clrln (10,11);
      gotoxy(1,10);
      writeln     ('File umbenennen :',inharray(.arraynr.));
      line_editor ('Neuer Name
                  ',1,11,8,filename,(.'0'..'9','A'..'z'.));
      Umw_in_Grossbuchstaben (filename);
      for i:= 1 to arraylaenge do
          if filename = inharray (.i.) then file_exist:= true;
      if file_exist then
         begin
         gotoxy(1,11);clreol;
         write (bell,'ein File ',filename,' existiert bereits.
                Taste drücken ');
         repeat until Keypressed;
         end (* if file-exist *)
       else begin
            assign (filevar,inharray (.arraynr.)+'.chn');
            rename (filevar,filename+'.chn');
            inharray (.arraynr.):= filename;
            init_array;
            clrln (17,24);
            schreib_array (inharray,arraylaenge,1,17,15,4);
            end; (* else *)
      end; (* if arraynr <> 0 *)
   clrln (9,11);
   end; (* case umbenennen *)
5: begin (* Rückkehr zu CP/M *)
   clrscr; write ('CP/M - Betriebssystem');
   writeln;writeln;
   halt;
   end; (* case CP/M *)
 end; (*case *)
 until true=false;
 end.
```

Listing. »filer.pas« (Schluß)

```
(* Library-include Proceduren *)
function exist(filename: anystring): boolean;
(* prüft, ob File auf Diskette vorhanden  *)
(* wenn vorhanden wird exist=true gesetzt *)
var f: file;
begin
   (*$I-*)
   assign(f,filename);
   reset(f);
   (*$I+*)
   if IOresult<>0 then exist:=false
   else exist:=true;
end; (* function exist *)
procedure Umw_in_Grossbuchstaben (var text:anystring);
(*$V-*)
var  i: integer;
begin
for i:= 1 to length(text) do
    text(.i.):= upcase (text(.i.));
end; (* procedure Umw_in_Grossbuchstaben *)
procedure schreib_array (var name:                        defstring;
                             anz,xstart,ystart,abstand,
                             worte_pro_Zeile:              integer);
(* schreibt array an vorgegebene Stelle des Bildschirms *)
(* defstring muss global definiert werden              *)
(* unter anz ist die Länge des Arrays zu verstehen     *)
var
i,x,y,wortzaehler : integer;
begin
x:= xstart; y:=ystart; wortzaehler:= 0;
gotoxy(xstart,ystart);clreol;
for i:= 1 to anz do
    begin
    gotoxy(x,y);
    write (name(.i.));
    x:= x+abstand;
    wortzaehler:=wortzaehler + 1;
    if wortzaehler= worte_pro_zeile
    then begin
         y:= y+1; x:= xstart; wortzaehler:= 0;
         gotoxy(x,y); clreol;
         end;
    end; (* for i *)
end; (* procedure schreib_array *)

function select  (var name                        : defstring;
                      anz,xstart,ystart,abstand,
                      worte_pro_zeile              : integer): integer;
(* Bewegt das Leuchtfeld entsprechend der definierten Tasten    *)
(* wird die ESC-Taste gedrückt, so wird select=0 zurückgeliefert *)
(* ansonsten die lfdnr des übergebenen Arrays                   *)
const
(* Diese Codes beziehen sich auf das Terminal *)
(* Basis 108 und müssen auf andere Terminals  *)
(* angepasst werden.                          *)

hoch =  #139;  (* Leuchtfeld 1 Zeile nach oben  *)
runter= #138;  (* Leuchtfeld 1 Zeile nach unten *)
rechts= #149;  (* Leuchtfeld 1 Wort nach rechts *)
links=  #136;  (* Leuchtfeld 1 Wort nach links  *)
bell  = ^G;    (* Glocke                        *)
esc   = #27;   (* ESC - Taste                   *)
home  = #192;  (* HOME - Taste                  *)
var
zch             : char;
x,y,xmax,ymax,i,
arraynr         : integer;
on_off          : boolean;

procedure invers_on_off;
begin
gotoxy(x,y);
for i:= 1 to abstand do
   write (' ');
on_off:= not (on_off);
if on_off then lowvideo;
gotoxy(x,y);
write (name(.arraynr.));
normvideo;
end; (* procedure invers_on_off *)
(* Hauptprogramm selector *)
begin
x:= xstart; y:= ystart; arraynr:= 1; on_off:= false;
xmax:= worte_pro_zeile * abstand;
ymax:=( anz div worte_pro_zeile)+ystart;
if anz mod worte_pro_zeile > 0 then ymax:=ymax + 1;
invers_on_off;
repeat
   repeat
   read (kbd,zch);
   until zch IN (.hoch,runter,rechts,links,esc,home.);
case zch of
hoch: begin
      invers_on_off;
      y:=y-1;
      if y<ystart then
         begin
         y:= ystart;
         invers_on_off;
         end (* if *)
      else begin
           arraynr:= arraynr - worte_pro_zeile;
           invers_on_off;
           end;
      end; (* case hoch *)
```

Listing. »lib.inc«

```
runter: begin
        invers_on_off;
        y:=y+1;
        if y>ymax then
           begin
           y:= y-1;
           invers_on_off;
           end (* if *)
        else begin
             arraynr:= arraynr + worte_pro_zeile;
             if arraynr> anz then
                begin
                arraynr:= arraynr-worte_pro_zeile;
                y:=y-1;
                end;
             invers_on_off;
              end; (* else *)
         end; (* runter *)
rechts: begin
        invers_on_off:
        x:=x+abstand;
        if x>xmax then
           begin
           x:= x-abstand:
           invers_on_off;
           end
        else begin
             arraynr:= arraynr + 1;
             if arraynr > anz then
                begin
                arraynr:= anz;
                x:= x-abstand;
                end;
              invers_on_off;
             end: (* else *)
        end; (* rechts *)
links : begin
        invers_on_off;
        x:=x-abstand:
        if x<xstart then
           begin
           x:= xstart;
           invers_on_off;
           end
        else begin
             arraynr:= arraynr - 1;
             if arraynr<1 then
                begin
                 arraynr:= 1;
                 x:= xstart:
                 end:
              invers_on_off;
             end: (* else *)
        end; (* links *)
esc : begin
      invers_on_off;
      arraynr:= 0;
      zch:= home;
      end; (* esc *)
end: (* case *)
until zch=home;
if arraynr<> 0 then invers_on_off;
select:= arraynr;
end; (* function select *)
procedure line_editor (text           : anystring;
                       xstart,ystart,
                       eingfeld       : integer;
                       var stringvar  : anystring:
                       addstring      : defset):
(* Wird die ESC-Taste gedrückt, so wird ein leerer String    *)
(* zurückgeliefert. Unter eingfeld ist die Länge des zurück- *)
(* zuliefernden Strings zu verstehen.                        *)
(*$V-*)
const  (* Diese Codes überprüfen und evtl. anpassen    *)
       bell    = ^G:    (* Klingelzeichen             *)
       Esc     = #27:   (* ESC-Taste                  *)
       return  = #13:   (* Return Taste               *)
       backspc = ^H:    (* Cursor 1 Zeichen nach links *)

var    x,i,
       zaehler  : integer;
       zch      : char;
begin
zaehler:= 0; stringvar:=''; x:= length (text) +3;
(* Schreibe Text und stelle Cursor auf  *)
(* Eingabefeld                          *)
gotoxy (xstart,ystart);clreol; write (text,' :');
for i:= 1 to eingfeld do
       write (' '):
write (':');
gotoxy(x,ystart);

(* Erzeugen der auszugebenden Variablen *)
repeat (* until return *)
  repeat
  read (kbd,zch):
  until (zch IN (.backspc,return,esc.)) or (zch IN (addstring));
if (zaehler >= eingfeld) and (not(zch IN (.backspc,return,esc.))) then
   write (bell)
else begin
     case zch of
     backspc: begin
              if length(stringvar)>0 then
                 begin
                 delete (stringvar,length(stringvar),1);
                 write (backspc,' ',backspc);
                 zaehler:= zaehler-1;
                 end;
               end; (* backspc *)
     Esc: begin
          stringvar:= '';
          zch:= return;
          end; (* esc *)
     return:;
     else
     stringvar:= stringvar+zch;
     write (zch);
     zaehler:= zaehler + 1;
     end: (* case *)
     end; (* else *)
     until zch = return;
     end; (* procedure line_editor *)
     procedure clrln (start,ende:integer);
     (* löscht die Zeilen von Start bis Ende *)
     var  i: integer;
     begin
     for i:= start to Ende do
         begin
         gotoxy(1,i);clreol;
         end;
     end; (* procedure clrln *)
     (* Ende der Library-include Proceduren *)
```

Listing. »lib.inc« (Schluß)

```
     (* Die Benutzer-Programme xxx.chn sollten    *)
     (* mit dieser Procedur abgeschlossen werden, *)
     (* um nach Filer zurückkehren zu können      *)

     procedure ende;
     var   diskfile : file;
     begin
     assign (diskfile,'filer.com');
     execute (diskfile);
     end;
```

Listing. »ende.inc«



Zeichenfolge analysiert

Nicht nur zum Compilerbau müssen Zeichenketten untersucht werden. Wie man so etwas macht, das lesen Sie hier.

Formale Sprachen, wie zum Beispiel alle Programmiersprachen, sind künstliche Sprachen. Sie werden jedoch nach den gleichen Gesetzen wie unsere natürlichen Sprachen gebildet. Dazu muß man sich über einige Grundbegriffe Klarheit verschaffen.

Vokabular = Menge der existierenden Wörter
Satz = Folge von Wörtern
Syntax (Grammatik) = Regel zur Bildung von korrekten Sätzen

Die Syntax legt fest, welche Wortfolgen zu einer Sprache gehören. Genau dies muß auch beim Compilerbau beachtet werden, da der Compiler beim Übersetzen überprüfen muß, ob ein Programm syntaktisch richtig geschrieben ist.

Die beiden nachfolgenden Programme lösen diese Aufgabe für eine sehr einfache formale Sprache.

Gegeben ist folgende Grammatik:
<A> : : = a (<A>)
Damit gelten zum Beispiel folgende Sätze:
<A> → ((a))
<A> → a
<A> → (((a)))

Das bedeutet, daß unsere Sprache nur drei Zeichenelemente kennt »a«, »(« und »)«. Das Bildungsgesetz wird aus den Beispielen oben deutlich: entweder handelt es sich um das Zeichen »a«, oder, falls Klammern gesetzt sind, muß die Anzahl der linken Klammern ebenso groß sein wie die der rechten.

Lösung 1: rekursive Analyse

Die Aufgabe wird in dem ersten Listing mit Hilfe einer rekursiven Prozedur gelöst. In der Prozedur wird jede öffnende Klammer auf dem Stack abgelegt. Ist das »a« erreicht, sozusagen der »Wendepunkt«, wird durch die schließenden Klammern der Stack wieder geleert.

Falls an irgendeiner Stelle ein Fehler festgestellt wurde, so setzt der Computer ein »Fehlerflag« (FEHLER : BOOLEAN).

Die Grundstruktur zur Feststellung des übergebenen Zeichens stellt ein CASE-Konstrukt dar, in dem vier Fälle unterschieden werden:

1. Zeichen = »(«
2. Zeichen = »a«
3. Zeichen = »)«
4. alle anderen Zeichen (OTHERWISE)

Im ersten Fall müssen zwei grundsätzliche Aufgaben gelöst werden: a) Rekursiver Prozeduraufruf mit einem eben gelesenenen Zeichen und b) nach der Rückkehr von der höheren Rekursionsebene die Überprüfung, ob zu der öffnenden Klammer der Prozedurebene das nächste Zeichen, welches eine schließende Klammer sein sollte, paßt. Im Fehlerfall wird an den Variablenparameter FEHLER (Fehlerflag) die »Falsch«-Information weitergegeben. Im zweiten Fall bleibt, da »a« ein erlaubtes Zeichen ist, aber danach nur mehr das Leeren des Stacks stattfindet, der Anweisungsteil leer. Der dritte Fall entspricht wiederum dem ersten und der vierte und letzte Fall ist sehr einfach, da hier nur das Fehlerflag gesetzt werden muß (unzulässiges Zeichen).

Im Hauptprogramm werden nach dem Bildschirmaufbau und der Eingabeaufforderung zunächst durch den Konstrukt

```
REPEAT
READ(ZEICHEN);
UNTIL ZEICHEN <> ' ';
```

führende Leerzeichen überlesen, wodurch nach dem Verlassen der Schleife in ZEICHEN schon das erste Zeichen steht. Als nächstes wird das Fehlerflag mit FALSE initialisiert. Anschließend kann nun die Untersuchung des ersten eingelesenen Zeichens in einem CASE-Konstrukt folgen.

Für den Fall einer öffnenden Klammer wird sofort mit diesem Zeichen und dem Fehlerflag die Prozedur aufgerufen, die sich wiederum selbst so oft aufruft, bis das aktuelle Zeichen ein »a« ist. Danach werden die auf den Stack abgelegten öffnenden Klammern wieder abgetragen. Ist das Zeichen ein »a«, passiert nichts, denn das Ganze ist schon fast fertig.

```
program zeichen_rekursiv (input,output);

(******************************************)
(***                                    ***)
(*** Rekursive Analyse einer Grammatik ***)
(***                                    ***)
(******************************************)

var zeichen : char;
    fehler  : boolean;

procedure a (z : char; var fehler : boolean);

(************************************)
(***                              ***)
(*** Rekursive Analyse Precedure ***)
(***                              ***)
(************************************)

begin
   case z of
      '(' : begin
               read(z);
               a(z,fehler);
               read(z);
               if (z <> ')') then fehler := true;
            end;
     'a' : ;
     otherwise
        fehler := true;
   end; (* case z of *)
end; (* Procedure a *)

begin

(**********************)
(***              ***)
(*** Hauptprogramm ***)
(***              ***)
(**********************)

   writeln;
   writeln (' Eingabe ');
   readln;
   repeat
      read(zeichen);
   until (zeichen <> ' ');
   fehler := false;
   case zeichen of
      '(' : a (zeichen,fehler);
      'a' : ;
      otherweise
         fehler := true;
   end; (* case zeichen of *)
   read(zeichen);
   if (zeichen <> ' ') then
      fehler := true;
   writeln;
   if fehler
      then
         writeln ('* Fehler *')
      else
         writeln ('# O.K. #');
   writeln;
end.
```

Listing 1. Rekursion ist die elegante Lösung

Lediglich wenn noch ein anderes Zeichen gelesen wurde, wird ein Fehlerfall durch das Fehlerflag angezeigt. Um auszuschließen, daß durch ein noch folgendes Zeichen das Eingabewort noch unerkannterweise falsch wäre, wird dies eingelesen und überprüft: ist es ungleich einem Leerzeichen, wird wiederum das Fehlerflag aktiv. Den Schluß bildet noch ein IF-THEN-ELSE-Konstrukt, das dem Benutzer aufgrund der Variablen FEHLER mitteilt, ob das Eingabewort syntaktisch korrekt ist oder nicht.

Lösung 2 : Iterative Analyse

Dieses Programm löst die Syntaxüberprüfung auf »konventionellem« Wege. Dazu werden zunächst wie bei der rekursiven Lösung erst der Bildschirmaufbau, Überlesen führender Leerzeichen und die Eingabe erledigt. Danach beginnt die eigentliche Analyse. Um gleich zu Beginn Fehlerfälle auszuschließen, entscheidet eine Abfrage, ob das erste gefundene Zeichen erlaubt ist, also entweder »a«, »(« oder »)«. Im Fehlerfall wird das Fehlerflag gesetzt. Falls es sich um ein erlaubtes Zeichen handelte, kann in einem WHILE-Konstrukt der Klammerebenenzähler solange inkrementiert werden, wie das nachfolgend eingelesene Zeichen ein »(« ist. Damit steht in dem Zähler die Zahl der öffnenden Klammern. Nun wird abgefragt, ob das von der Klammer verschiedene Zeichen ein »a« ist. Das Fehlerflag zeigt einen Fehler an. Im anderen Fall kann mit dem Zählen der schließenden Klammern begonnen werden. Dazu dient wieder ein WHILE-Konstrukt mit Dekrementierung der Klammerebene und dem Einlesen des nächsten Zeichens.

In der nun letzten Abfrage, ob das nachfolgende Zeichen ein Leerzeichen oder die Klammerebene gleich Null ist, werden die letzten Fehlermöglichkeiten des Wortes abgefangen.

Zum Schluß wird wieder ebenso wie in der rekursiven Lösung, je nachdem, ob ein Fehler festgestellt wurde oder nicht, ein positiver oder negativer Hinweis ausgegeben.

(Max Moser/Harry Paintner/hg)

```
program zeichenfolgen_iterativ (input,output);

(****************************************)
(***                                  ***)
(*** Iterative Analyse einer Gramatik ***)
(***                                  ***)
(****************************************)

var zeichen : char;
    fehler  : boolean;
    ebene   : integer;

(*********************)
(***             ***)
(*** Hauptprogramm ***)
(***             ***)
(*********************)

begin
   writeln;
   writeln (' Eingabe ');
   readln;
   repeat
      read(zeichen);
   until (zeichen <> ' ');
   fehler := false;
   if (zeichen <> '(') and (zeichen <> 'a')
      then
         fehler := true
      else
 begin
    ebene := 0;
    while (zeichen ='(') do
       begin
          ebene := ebene + 1;
          read (zeichen);
       end;
    if (zeichen <> 'a')
       then
          fehler := true
       else
          begin
             read(zeichen);
             while (zeichen = ')') do
                begin
                   ebene := ebene - 1;
                   read (zeichen);
                end;
             if (zeichen <> '  ') or (ebene <> 0) then
                fehler := true;
          end;
 end;
 writeln;
 if fehler
    then
       writeln('* Fehler *')
    else
       writeln('# O.k. #');
 writeln
 end.
```

Listing 2. Die iterative Umsetzung ist nicht ganz so schön

Rekursive Spielereien

Die Fibonacci-Zahlen sind ein schönes Beispiel, um die Unterschiede zwischen rekursiver und iterativer Programmierung verstehen zu lernen.

Ein lehrreiches Beispiel der rekursiven Programmierung, das heißt der Programmierung von Unterprogrammen, die sich selbst aufrufen, ist ein Lösungsweg zur Berechnung der Fibonacci-Zahlen.

Der im Mittelalter lebende, italienische Mathematiker Leonardo Pisano (zirka 1180 bis 1240), stellte eine Zahlenfolge auf, in der jede Zahl die Summe der beiden vorhergehenden Zahlen darstellt und deren erste Elemente mit 0 und 1 vordefiniert sind.

Daraus ergeben sich folgende Elemente der Folge:

```
0. Element :  0
1. Element :  1
2. Element :  1 =  0 + 1
3. Element :  2 =  1 + 1
4. Element :  3 =  2 + 1
5. Element :  5 =  3 + 2
6. Element :  8 =  5 + 3
7. Element : 13 =  8 + 5
8. Element : 21 = 13 + 8 und so weiter
```

Man kann die Entwicklung auch mit folgender Formel darstellen:

f(i) = f(i-1) + f(1-2)

Es gibt nun zwei Möglichkeiten, die Fibonacci-Zahlen mit dem Computer iterativ und rekursiv zu bestimmen:

Listing 1 zeigt die iterative Lösung mit einer REPEAT-UNTIL-Anweisung. Sie ist zwar ebenso effektiv wie die rekursive Lösung (Listing 2), aber bei weitem nicht so elegant.

Allerdings ist zu beachten, daß bei rekursiver Programmierung der Stack überlaufen kann, wenn man zu große Zahlenwerte benutzt, da alle Daten der Rekursion (Rechenergebnisse, Rücksprungpointer und so weiter) auf dem Stack zwischengespeichert werden.

So wird zum Beispiel das 4. Element der Fibonacci-Folge in folgenden Schritten berechnet (siehe Bild).

(Max Moser/Harry Painter/hg)



```
            FIBONACCI(4)
          = FIBONACCI(3)                              +FIBONACCI(2)
          = FIBONACCI(2)                +FIBONACCI(1)+FIBONACCI(1)+FIBONACCI(0)
          = FIBONACCI(1)+FIBONACCI(0) +1              +1           +0
          = 1           +0            +2
          = 3
```

Bild. Berechnung des 4. Elements der Fibonacci-Folge

```
program fibonacci_iterativ (input,output);
var index, zahl, vletzte, letzte, aktuell : integer;
begin
   repeat
      write ('Bitte geben Sie die Elementnummer der Fibonacci-Folge ein: ');
      readln (zahl);
   Until Zahl >= 0;
   write ('Die Fibonacci-Zahl ist ');
   if zahl <= 2 then
   begin
      case zahl of
         0   : writeln ('0');
         1,2 : writeln ('1');
      end;
   end else
   begin
      vletzte := 1;
      letzte  := 1;
      index   := 2;
      repeat
         aktuell := vletzte + letzte;
         vletzte := letzte;
         letzte  := aktuell;
         index   := index + 1;
      until index = zahl;
      writeln (aktuell);
   end;
end.
```

Listing 1. Die Fibonacci-Zahlen iterativ berechnet

```
program fibonacci_rekursiv (input,output); (*$A-*)
var zahl : integer;
function fibo (wert : integer ) : integer;
begin
   if wert > 1
      then fibo := fibo (wert-1) + fibo (wert-2)
         else if wert = 1
         then fibo := 1
         else fibo := 0
end;

(* Hauptprogramm *)

begin
   repeat
      write ('Bitte geben Sie die Elementnummer der Fibonacci-Folge ein: ');
      readln (zahl);
   until zahl >= 0;
   writeln ('Die Fibonacci-Zahl ist ',fibo (zahl));
end.
```

Listing 2. Die Fibonacci-Zahlen rekursiv berechnet. Die Zeichenfolge »(*$A- *)« schaltet den Compiler auf rekursiven Code um. Viele Compiler (so auch Turbo-Pascal) arbeiten mit so einer Umschaltanweisung.

Numerische Integration nach Simpson

Integralberechnung ist mit dem Computer in Pascal eine feine Sache. Nur die richtige Formel muß man kennen.

Für mathematische Berechnungen sind Computer geradezu prädestiniert, man muß ihnen nur sagen, wie. Mit der Simpson-Formel lassen sich bestimmte Integrale der Form

$$F(x)\Big|_a^b = \int_a^b f(x)dx$$

näherungsweise berechnen.
Die Voraussetzungen hierbei sind:
- a,b sind reelle Zahlen und es gilt $a < b$.
- $y=f(x)$ sei eine im Intervall [a,b] stetige und differenzierbare Funktion.

Die Lösungsformel lautet:

$$F(a,b,2n) = (h/3) * [f(a) + f(b) + 4 \sum_{i=1}^{n} f(a + (2i-1)h) + 2 \sum_{i=1}^{n-1} f(a + 2ih)]$$

wobei das Intervall [a,b] in 2n Teilintervalle der Breite
$h = (b - a)/2n$
geteilt wird.
Das folgende Programm berechnet mit Hilfe der Funktion SIMPSON zu

der gegebenen Funktion $f(x) = \exp(-x^2) \int_0^x \exp(t^2)dt$ mit
dem Intervall [0,2]
und der Stellengenauigkeit m=5
die Näherungswerte $F(a,b,2^i)$ mit $1 <= i <= k$.

Die Berechnung wird abgebrochen, wenn folgendes Kriterium erfüllt ist:
$[F(a,b,2^k) - F(a,b,2^{(k-1)})] <= |F(a,b,2^k)| * 0,5 * 10^{-m}$.

Programmbeschreibung:

In der Darstellung (Bild 1) sei die Unterteilung des Intervalls, sowie die Gewichtung der Funktionswerte an den Stützpunkten in der Simpson-Formel verdeutlicht:

Das Integral ergibt sich daraus folgendermaßen:
I = (h/3)*(Fktwert(a)+Fktwert(b)+4*(Fktwert(S1)+Fktwert(S3)+2* Fktwert(S2)

Da bei dieser Intervallunterteilung noch keine großen Ansprüche an die Rechengenauigkeit gestellt werden können, nimmt man eine feinere Intervalleinteilung vor.

Hierbei benutzen wir folgenden Trick:

Die Anzahl der Teilintervalle wird nicht um zwei erhöht, sondern verdoppelt (Teilintervallhalbierung).

Der Schritt der Teilintervallhalbierung und seiner Auswirkung ist in Bild 2 zu sehen.

Dies hat folgende Vorteile:
- Aus der Tatsache, daß die bisherigen Stützpunkte wieder benutzt werden, folgt eine wesentliche Verminderung des Rechenaufwandes.
- Da die Anzahl der Teilintervalle exponentiell ansteigt, konvergiert dieses Verfahren sehr schnell.

Dabei fällt folgendes auf:
- Alle bisherigen Stützpunkte, sowohl die mit der Gewichtung 2 als auch die mit der Gewichtung 4, erhalten bei diesem Schritt die neue Gewichtung 2.
- Die Anzahl der neuen Punkte (also die mit der Gewichtung 4) ist auf das Doppelte gewachsen.
- Die neuen Stützpunkte sind jeweils um die doppelte Intervallbreite voneinander entfernt.

Um diese Erscheinung nochmals zu verdeutlichen, ist in der Darstellung in Bild 3 eine weitere Intervallhalbierung vorgenommen worden.

Durch die stetige Intervallhalbierung wird irgendwann eine genügende Genauigkeit erreicht.

Zur Feststellung dieser Genauigkeit beziehungsweise als Abbruchkriterium für die Unterteilungsschleife eignet sich eine Formel, worin 2^k beziehungsweise 2^{k-1} zwei aufeinanderfolgende Intervallhalbierungen bedeuten. Dieses Abbruchkriterium verwendet schließlich auch M zur Angabe der Zahl der wesentlichen Stellen.

Die bisherigen Erläuterungen des Verfahrens lassen sich einfach in einen Algorithmus umsetzen.

Beim Funktionsaufruf werden eine reelle Funktion f, die untere und obere Integrationsgrenze (--> ug und og, reell), sowie die Rechengenauigkeit m (integer) angegeben. Das Ergebnis der Integration ist natürlich wieder reell.

Damit sieht der Funktionskopf wie folgt aus:

```
FUNCTION SIMPSON (FUNCTION F(X : REAL) : REAL;
                                OG, UG : REAL;
                                M      : INTEGER)
                                       : REAL;
```

In der Variablendeklaration sind folgende Bezeichner festgelegt:

```
SUMUO                   : REAL
```

- Summe der Fktwerte der Integrationsgrenzen

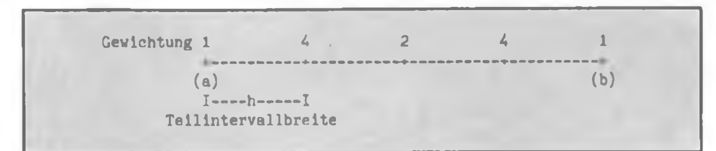


Bild 1. Gewichtung in Teilintervallbreite: zwei wichtige Faktoren für die Simpson-Formel

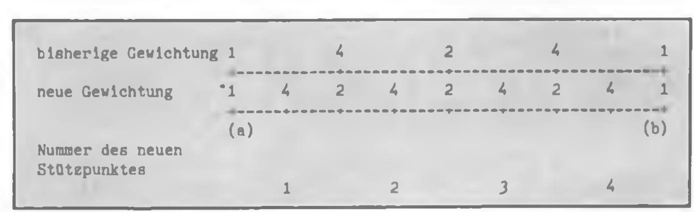


Bild 2. Verkleinern wir die Teilintervalle, so wird das Ergebnis genauer

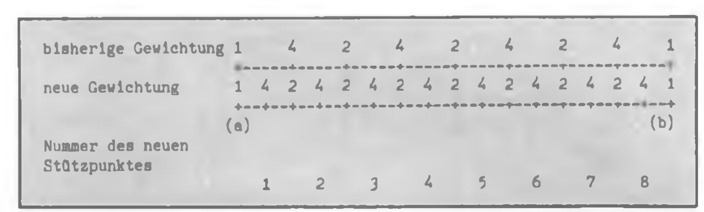


Bild 3. Jede weitere Halbierung macht das Ergebnis brauchbarer

SUMALT : REAL
- Summe der Fktwerte mit Gewichtung 2 in der Simpson-Formel
SUMNEU : REAL
- neuberechnete Summe der Fktwerte mit Gewichtung 4
INTALT, INTNEU : REAL
- letzter und aktueller Integralwert
DELTA : REAL
- aktuelle Teilintervallbreite (h)
ZNEUPKT : INTEGER
- aktuelle Anzahl der neuzuberechnenden Fktwerte
COUNT : INTEGER
- Schleifenzähler zur Summierung der Fktwerte
ZEHNHOCHM : INTEGER
- »Konstante«, enthält 10 für Abdruckkriterium

Nun zum eigentlichen Algorithmus. Beim Eintritt in das Programm sind mehrere Größen zu initialisieren:
SUMUO : = F(UG) + F(OG)
- SUMUO wird für den ganzen Fktaufruf so festgelegt.
DELTA : = (OG - UG)/2
- Startwert der Teilintervallbreite
ZEHNHOCHM : = TRUNC(EXP(M*LN(10))) = 10↑M
SUMALT : = 0
INTNEU : = 0
SUMNEU : = 0
Startwerte für ersten Durchlauf
ZNEUPKT : = 1
- beim ersten Durchlauf nur 1 neuer Fktwert

Nach der Initialisierung folgt nun die »Integrationsschleife«. Ihre grundsätzliche Struktur stellt ein REPEAT-UNTIL-Konstrukt dar, wobei als Abbruchkriterium die eingangs erwähnte Formel dient.

In der Schleife werden folgende Aufgaben gelöst:
1.) Aufaddieren der neuen Funktionswerte (Abstand: doppelte Teilintervallbreite mit Beginn beim 1. Stützpunkt nach der Untergrenze).
2.) Wenn das Abbruchkriterium nicht erfüllt ist:
a.) Verdoppeln der Anzahl der neu zu berechnenden Stützstellen = ZNEUPKT
b.) Halbieren der Intervallbreite
c.) Sichern des berechneten Integrals für den Vergleich beim nächsten Schleifendurchlauf.
d.) Umbenennen der Funktionswerte mit Gewichtung 4 zu den Funktionswerten mit Gewichtung 2
e.) Neuinitialisierung der Summe mit Gewichtung 4 auf 0

In der Funktion SIMPSON sieht die Schleife wie folgt aus:

```
REPEAT
 INTALT := INTNEU;
 SUMALT := SUMALT + SUMNEU;
 SUMNEU := 0;
 FOR COUNT := 1 TO ZNEUPKT DO
SUMNEU := SUMNEU + F(UG+(2*COUNT-1)*DELTA);
 ZNEUPKT := ZNEUPKT*2;
 DELTA := DELTA/2;
UNTIL (Abbruchkriterium erfüllt);
```

Den letzten Teil der Funktion bildet schließlich die Übergabe des beim letzten Durchlauf berechneten Integrals, bei dem das Abbruchkriterium erfüllt wurde, an den Funktionsbezeichner.

Sollte der zur Verfügung stehende Compiler die Übergabe von Funktionen an eine Funktion nicht erlauben (zum Beispiel Turbo-Pascal), so muß man die Funktion SIMPSON im Hauptprogramm realisieren oder man behilft sich, indem man die

Funktion als global vereinbart. Sie muß dann aber vor jedem Programmstart neu eingegeben werden – der Bedienungskomfort sinkt.

Um die Funktion SIMPSON überhaupt nutzen zu können, benötigt man noch ein Rahmenprogramm, das die zusammengesetzte Funktion f(x) im Intervall [a,b] an n+1 Stützstellen auf die wesentliche Stellengenauigkeit m tabelliert.

Die Platzhalter der Daten für Intervallanfang, -ende, Stellengenauigkeit und Stützpunkte wurden als Konstante dargestellt. Sie könnten jedoch auch ohne weiteres vom Programm eingelesen werden. Somit sieht der Programmkopf wie folgt aus:

```
PROGRAM RAHMEN (INPUT;OUTPUT);
 CONST A = 0.0; (*Tabellierungsanfang *)
       B = 2.0; (*Tabellierungsende *)
          N =  40; (*N+1 Stützstellen zur
                    Tabellierung *)
          M =   5; (* Anzahl der signifikanten
                    Stellen zur Integration *)
```

```
VAR  X : REAL; (* Stützstelle als Argument für die
                  Funktion *)
     H : REAL; (* Schrittweite zur Tabellierung *)
     I : 0 .. N; (* Schleifenzähler für die
                  Stützstellen *)
```

Darauf folgt der Funktionsdeklarationsteil. Zum einen wird eine Funktion benötigt, die die angegebene, zusammengesetzte Funktion beinhaltet (FUNCTION FKT). In dieser Funktion ist auch die Integrandenfunktion TEILFKT eingebettet, die dann als aktuelle Parameterfunktion für die Integrationsfunktion SIMPSON verwendet wird. Hier wird mit dem linken Term EXP(-X*X) das Integral von TEILFKT EXP(X*X) in den Grenzen von UG bis OG auf eine Genauigkeit von M wesentlichen Stellen berechnet.

Im abschließenden Hauptprogramm findet nur noch die Berechnung der Funktionswerte an den Stützstellen, sowie die tabellarische Ausgabe der Ergebnisse statt.

(Max Moser/Harry Paintner/hg)

Structogramm der Funktion Simpson

- Initialisieren der Summe des Anfangs- und Endwerts der Intervallbreite des Integrals, der Hilfsvariablen für die neuen bzw. alten Integralwerte und der Hilfsvariablen für die Summe der neuen bzw. alten Stützpunkte des Integrals und des Anfangswerts der Anzahl der Teilintervalle des Integrals.
- Teilintervallbreite des Integrals := (Obergrenze-Untergrenze) / 2
- Hilfsvariable für Zehnerpotenz := EXP (Stellengenauigkeit * LN(10))
- (Wiederholen:)
  - Sicherung des letzten Integralwerts für die Abbruchentscheidung
  - Sicherung der Summe der letztberechneten Stützpunkte
  - Integralvariable auf Null setzen
  - Schleife von 1 bis Anzahl Stützpunkte
    - Aufaddieren der neuen Stützpunkte
  - Berechnung des neuen Integralwerts
  - Verdoppeln der Anzahl der Stützpunkte
- Wiederholen des Blocks, bis die gewünschte Stellengenauigkeit erreicht ist.

Structogramm der Funktion FKT

- Konstante für Integraluntergrenze
- Funktionsaufruf SIMPSON, die Funktion TEILFKT, Integraluntergrenze, Argument und Stellengenauigkeit werden als Aktualparameter übergeben
- Funktionswert FKT := EXP (-x * x) * Funktionswert von SIMPSON

Structogramm des Hauptprogramms

- Konstanten für Intervallanfang, Intervallende, Anzahl Teilintervalle und die gewünschte Stellengenauigkeit
- Ausgabe der Überschrift für die Funktionsliste
- Teilintervallbreite := (Intervallende -Intervallanfang) / Anzahl der Teilintervalle
- Schleife von 0 bis Anzahl der Teilintervalle
  - Argument := Intervallanfang + Schleifenindex * Teilintervallbreite
  - Funktionsaufruf von FKT, Argument und Stellengenauigkeit werden als Aktualparameter übergeben
  - Ausgabe des Arguments und des Funktionswerts in die Ausgabeliste

```
program rahmen (input,output);

(*****************************************************************)
(***                                                          ***)
(*** Rahmenprogramm zur Naeherungsweisen Integration nach Simpson ***)
(***                                                          ***)
(*****************************************************************)

const a = 0.0;
      b = 2.0;
      n =  40;
      m =   5;
var   x,h:real;
      i  :0 .. n;

(*****************************************************************)
(***                                                          ***)
(***     Naeherungsweise Integration ueber eine Funktion f(x)     ***)
(***                                                          ***)
(*****************************************************************)

function simpson (function f(x:real):real;
                           ug,og    :real;
                           m        :integer):real;

var sumuo,sumalt,sumneu       : real;
    intalt,intnew,delta       : real;
    zneupkt,count,zehnhochm   : integer;

begin
   sumuo     := f(ug)+f(og);
   delta     := (og-ug)/2;
   zehnhochm := trunc (exp(m*ln(10)));
   sumalt    := 0;
   sumneu    := 0;
   intneu    := 0;
   zneupkt   := 1;
   repeat
      intalt := intneu;
      sumalt := sumalt + sumneu;
      sumneu := 0;
      for count := 1 to zneupkt do
         sumneu := sumneu + f(ug+(2 * count-1) * delta);
      intneu := delta * (sumuo + 4 * sumneu + 2 * sumalt) / 3;
      zneupkt := zneupkt * 2;
      delta   := delta / 2;
   until (abs (intneu-intalt)*zehnhochm <= abs (intneu / 2));
   simpson := intneu;
end; (* of function simpson *)

(*****************************************************************)
(***                                                          ***)
(***                 Zu tabellierende Funktion                ***)
(***                                                          ***)
(*****************************************************************)

function fkt (x:real;m:integer) : real;
   const ug = 0.0;
   function teilfkt (t:real) : real;
   begin
      teilfkt := exp (t*t);
   end; (* of function teilfkt *)
   begin
      fkt := exp (-x * x) * simpson (teilfkt,ug,x,m);
   end; (* of function fkt *)

(*****************************************************************)
(***                                                          ***)
(***                       Hauptprogramm                      ***)
(***                                                          ***)
(*****************************************************************)

begin
   writeln; writeln; writeln;
   writeln ('     ***    x   ***          f(x)          ***  ');
   writeln;
   h := (b-a)/n;
   for i := 0 to n do
      begin
         x := a + i * h;
         writeln ('         ',x:12,'          ',fkt(x,m):16);
      end;
   writeln;
end.
```

Listing. Integration nach Simpson

Die Türme von Hanoi

Haben Sie auch Probleme mit der Lösung der Knobelaufgabe »Türme von Hanoi«? Aber benutzen Sie doch einfach Ihren Computer und lassen Sie ihn für sich knobeln.

Wer kennt nicht das Problem der Türme von Hanoi? Die Aufgabe: Es existieren drei Plätze, wobei auf einem der äußeren Standorte ein aus mehreren, immer kleiner werdenden Scheiben bestehender Turm steht. Nun muß dieser Turm auf den anderen äußeren Platz umgelagert werden. Natürlich soll nicht der komplette Turm auf einmal transportiert werden, sondern Scheibe für Scheibe. Bei dieser Umschichtung darf immer nur eine kleine Scheibe auf eine größere Scheibe gelegt werden.

An dieser Knobelaufgabe haben sich schon große Geister versucht und auch sehr lange darüber gegrübelt. Wenn nur drei Scheiben auf dem Turm liegen, ist es ja noch relativ einfach herauszufinden. Haben Sie dagegen einen Turm mit vielleicht 20 Scheiben, so ist es so gut wie nicht mehr zu schaffen (es sei denn, Sie hätten einige Jahre Zeit dafür).

Die Prozedur »hanoi« (Listing) nimmt Ihnen nun die Denkarbeit vollständig ab. Sie müssen nur die Anzahl der Scheiben des Turmes eingeben, und schon erscheint die Ablegevorschrift auf dem Bildschirm.

Versuchen Sie aber ruhig zuerst einmal, das Problem mit drei, vier oder fünf Scheiben »zu Fuß« anzugehen, bevor Sie sich die Lösung vom Computer zeigen lassen.

(Dieter Mayer/hi)

```
PROGRAM hanoi(INPUT,OUTPUT);
VAR
   n,i:real;
   etagenzahl:0..MAXINT;

   PROCEDURE verlagere(hoehe,von,zu,ueber:integer);
          PROCEDURE tragt(runter,rauf:integer);
                 BEGIN
                     write(runter:1,'>',rauf,'   ');
                     i:=i+4;
                     n:=n+1;
                     IF i>=80 THEN BEGIN
                       i:=0;
                     END (*end if*)
                 END;
   BEGIN (*verlagere*)
         IF hoehe>0 THEN BEGIN
            verlagere(hoehe-1,von,ueber,zu);
            tragt(von,zu);
            verlagere(hoehe-1,ueber,zu,von);
         END;    (*end if*)
   END;   (*end verlagere*)

BEGIN
     write('Wegen Verstaendigungs');
     writeln('schwierigkeiten in Hanoi muss der ');
     write('Turm von Hanoi-Nord nach Hanoi-Sued ');
     writeln('verlegt werden.');
     writeln('Ein Zwischenlager gibt es in Hanoi-Mitte.');
     writeln('Beim Verlegen ist zu beachten:');
     writeln('-nur die jeweils oberste Etage kann ');
     write(' von einem Lager in ein anderes');
     writeln(' transportiert werden');
     write('-eine Etage darf nicht auf ');
     writeln('eine kleinere gelegt werden');
     writeln;
     write('Bitte die Anzahl der Etagen angeben: ');
     readln(etagenzahl);
     i:=0;
     n:=0;
     verlagere(etagenzahl,1,2,3);
     writeln;
     writeln('Anzahl der Umlagerungen: ',n:5:0);

END.
```

Zahlen-Tabellen

Ein nützliches Programm, das Ihnen auf einen Blick Zahlen und deren Quadrat- oder Kubik-Funktionen übersichtlich aufzeigt. Es gibt oft Situationen, in denen diese kleine Anwendung recht nützlich ist.

Diese kleine Routine zeigt Ihnen die Quadrat-, Kubik-, »Hoch 4«- und »Wurzel«-Werte einer Zahl in einer übersichtlichen Tabelle. So können Sie sich auf einfache Art eine Liste erstellen, die Ihnen die gebräuchlichsten Werte auf einen Blick liefert. Allerdings werden nur Zahlen bis 181 richtig verarbeitet (dies hängt mit dem Format der Real-Zahlen zusammen).

Zum Programm

Die Prozedur »clrscr« ist MS-DOS-spezifisch. Man kann sie auch weglassen, da hiermit nur der Bildschirm gelöscht wird. Wollen Sie Zahlen ausgeben, die größer als 21 sind, so ist nur die Bedingung in der Until-Schleife auf einen höheren Wert als 21 zu setzen (maximal 181), oder das Format der Zahl in der Variablendefinition in Real umzuwandeln, und die Ausgabeformatierung »zahl« in der Repeat-Schleife durch »write(zahl:4:0,...« zu ändern.

(Dieter Mayer/hi)

```
PROGRAM tabelle;
VAR
   zahl:integer;
   wurzel,quadrat,kubik,hoch4:real;
BEGIN
  zahl:=0;
  clrscr;
  write('Zahl:         Quadrat:           Kubik');
  writeln(':         Hoch 4:          Wurzel:');
    repeat
      quadrat:=zahl*zahl;
      kubik:=quadrat*zahl;
      hoch4:=quadrat*quadrat;
      wurzel:=sqrt(zahl);
      write(zahl:2,'              ',quadrat:5:0);
      write('             ',kubik:5:0);
      write('         ',hoch4:7:0);
      writeln('         ',wurzel:10:7);
      zahl:=zahl+1;
    until zahl>21
END.
```

C – wie Cäsar

Der römische Imperator war der mächtigste Mann seiner Zeit. C herrscht genauso in der Welt der Programmiersprachen. Lernen Sie die faszinierende Sprache kennen.

C ist eine noch relativ junge Programmiersprache, die Anfang der siebziger Jahre von Dennis M. Ritchie entwickelt wurde. Er arbeitete damals in den Bell Laboratories in den USA und stand vor dem Problem, ein komplettes Betriebssystem (und zwar Unix) in Assembler zu schreiben. Bis zu diesem Zeitpunkt kannte man keinen anderen Weg Betriebssysteme zu entwickeln. Die Mikroprozessoren waren noch sehr langsam und Speicherbausteine teuer.

Doch Dennis Ritchie waren die Nachteile von Maschinencode klar: Solche Programme sind nur schwer auf andere Mikroprozessoren zu übertragen. Selbst die Adaption auf andere Computer mit dem gleichen Prozessor führt zu kaum überwindbaren Schwierigkeiten. Außerdem sollte Unix alle vorherigen Betriebssysteme in Bezug auf Leistungsfähigkeit in den Schatten stellen. Das bedeutete aber auch, daß der Umfang des Programmcodes gigantisch werden mußte.

Ab einer Länge von mehreren KByte sind Assemblerprogramme praktisch nicht mehr überschaubar und enthalten mit Sicherheit versteckte Bugs (bösartige Programmierfehler), die oft erst nach jahrelangem Einsatz des Systems zum Vorschein kommen. Noch heute gibt es in CP/M 2.2 kleine Fehler, die nie beseitigt wurden. Und das, obwohl die erste Version von CP/M immerhin im Jahr 1973 erschien.

So sah sich Ritchie nach einer Programmiersprache um, die sich zur Systemprogrammierung eignet. Da sich aber alle »konventionellen« Sprachen der damaligen Zeit – zum Beispiel Fortran, Algol und Cobol – als unbrauchbar erwiesen, entschloß er sich, eine neue Sprache für seine Zwecke zu schreiben. Am Ende dieser Entwicklung stand C in der Form, in der wir es heute kennen. Und Ritchie hat tatsächlich das Meisterstück vollbracht, etwa 95 Prozent von Unix in C zu programmieren. Eine beeindruckende Leistung, denn »seine« Programmiersprache ist schnell genug, mehrere Benutzer zu bedienen und mehrere Prozesse (Programme) gleichzeitig zu bearbeiten.

Natürlich stammt nicht alles zu 100 Prozent vom Entwickler selbst. Er griff auch auf ältere Sprachen zurück. Die »Vorfahren« von C sind CPL, BCPL und B – Programmiersprachen, die heute fast niemand mehr kennt. CPL, die »Combined Programming Language«, wurde zu Beginn der sechziger Jahre als eine Gemeinschaftsarbeit der Universitäten von Cambridge und London entwickelt. Aber: Viele Köche verderben den Brei. Das Resultat war eine Monstersprache, die den meisten Programmierern zu schwerfällig und zu undurchsichtig war. Rettung nahte, als Martin Richards in Cambridge begann, sich für die Sprache zu interessieren. Er kürzte alles Entbehrliche heraus, und fertig war die neue Sprache BCPL, die »Basic Combined Programming Language«. Eines der heute bekanntesten Produkte in BCPL dürfte der Basic-Interpreter für die ST-Computer sein.

Zur Systemprogrammierung besser geeignet, weil noch knapper gehalten, ist B. Die Sprache mit dem kurzen Namen stammt von Ken Thompson, der sie im Jahr 1970 im Auftrag der Bell Laboratories entwickelte. B besitzt nur einen einzigen Datentypen, das Maschinenwort. Das war Dennis Ritchie dann aber doch zu wenig, und als er sich für Unix zwischen B und Assembler entscheiden sollte, wählte er die – damals erst theoretisch existierende – Sprache C. C baut auf B auf, ist aber entschieden Programmierer-freundlicher. Nicht zuletzt diese Eigenschaft dürfte erheblich zu ihrer Verbreitung beigetragen haben.

Warum C?

Heute ist C auf dem besten Weg, sowohl Hochsprachen – wie Pascal und Fortran – als auch die diversen Maschinensprachen zu verdrängen. Die Entwicklung geht inzwischen dahin, nur noch einen C-Compiler für neue Prozessoren und Betriebssysteme in Assembler zu schreiben und die weitere Software mit dem C-Compiler zu erzeugen. Bei 16-Bit-Prozessoren mit 8 MHz Taktfrequenz –wie dem Motorola 68000 im Atari ST oder dem Amiga von Commodore – kann man sich den unvermeidlichen Geschwindigkeitsverlust durchaus leisten.

C bietet Kontrollstrukturen, die mit denen in Pascal, dem »Musterschüler« der strukturierten Programmierung, durchaus auf eine Stufe zu stellen sind. Pascals CASE heißt in C SWITCH CASE, REPEAT UNTIL und WHILE nennen sich DO WHILE und WHILE. Auch die allseits beliebte FOR-Schleife ist – wenn auch mit geänderter Syntax – wiederzufinden. Andererseits hat der Programmierer in C alle Möglichkeiten, auch auf der untersten Hardware-Ebene, direkt mit den Peripheriegeräten, zu arbeiten.

Eine der schönsten Eigenschaften von C haben wir aber beinahe vergessen: »Portabilität« heißt das Zauberwort. Wer von einem Computer auf einen anderen umsteigt, kann praktisch die gesamte C-Software mitnehmen. Diese Kompatibilität der einzelnen C-Versionen erreicht keine andere Sprache, nicht einmal das vielgerühmte Pascal. Versuchen Sie einmal, ein Programm, das starken Gebrauch von den Fähigkeiten des Turbo-Pascals macht, auf MS-Pascal oder Pascal/MT+ zu übertragen. Von den Hunderten von Basic-Dialekten wollen wir gar nicht erst reden.

Was für Hobby-Programmierer gilt, trifft für Software-Firmen, die Programme kommerziell vertreiben, noch viel mehr zu. Ein großer Teil der Produktionskosten für Software läßt sich einsparen, wenn Programme in kürzester Zeit auf anderen Computern zum Laufen gebracht werden können. Die Software-Entwicklung wird damit von den Fortschritten der Hardware unabhängiger.

Derart motiviert, können Sie sich sicher nicht mehr dem Bann entziehen, der von C ausgeht. Es stellt sich nur noch ein »kleines« Problem, bevor Sie mit dem Programmieren beginnen können: Sie brauchen einen Compiler. Für MS-DOS-Computer gibt es inzwischen fast mehr C- als Pascal-Compiler. Im Bereich der Heim-Computer ist das Angebot deutlich geringer. Für den Sinclair Spectrum und den Schneider-CPC bietet HiSoft C-Compiler an (wobei die Schneider-Version erheblich leistungsstärker ist). Für den Commodore 64 gibt es einen Compiler von Data Becker. Allerdings ist der 6502 für die Implementation eines C-Compilers und für die Bearbeitung von C-Programmen denkbar ungeeignet. Die 6502-CPU ist technisch veraltet und bietet mit drei Registern (Akku, Indexregister X und Y) nicht genug für eine so Register-intensive Sprache wie C. Wer einen Commodore 128 hat, sollte sich deshalb entschließen, auf CP/M umzusteigen.

Unter CP/M nämlich ist das Angebot reichhaltig. Ein großer Teil der Compiler kann aber auf dem CPC 464 und CPC 664 von Schneider ohne Speichererweiterung nicht zum Laufen gebracht

werden. Am besten stehen somit wieder einmal die Besitzer des CPC 6128 da, die über CP/M 3.0 mit 61 KByte Speicherplatz herrschen. Neben dem für Normalsterbliche völlig überteuerten C-Compiler von Digital Research (jenseits der Schallmauer von 1000 Mark) tummeln sich in der Gegend um etwa 500 Mark mehrere Programme, zum Beispiel MI-C (auch auf 3-Zoll-Disketten lieferbar) oder auch BDS-C, dem selbst HiSoft in der Anleitung zum eigenen C-Compiler Lob zollt. Besonders interessant für C-Begeisterte mit kleinem Geldbeutel dürfte der Small C-Compiler sein, den Markt & Technik für 148 Mark herausgebracht hat (angepaßt für Schneider und Commodore).

Fast alle Compiler verlangen denselben C-Quellcode. Manche Eigenschaften, die von Ritchie und seinem Kollegen Brian W. Kernighan im C-Standardwerk »The C Programming Language« (»Programmieren in C«), festgelegt sind, wurden aber in diversen Compilern nicht implementiert. Oft fehlen Bitfelder und Fließkommazahlen. Um Sie nicht zu verwirren, gehen wir in diesem Kurs auf solche eher ausgefallenen Dinge nur am Rande ein.

Um C zu verstehen, sollten Sie die im Text angegebenen Programmbeispiele nach Möglichkeit nachvollziehen. Wie ein Programm compiliert und gestartet wird, das steht in Ihrem Handbuch. CP/M-Programme erwarten den C-Quellcode meist in Form einer Textdatei, die zuvor mit einem Texteditor erzeugt wurde. HiSoft-C verlangt zum Beispiel den Befehl »HC PROGRAM.C« und macht daraus ein COM-File mit dem Namen »PROGRAM.COM«. Andere CP/M-Compiler erzeugen oft nur einen Assembler-Quellcode, der dann mit einem Assembler, wie ASM.COM, MAC.COM oder MACRO-80, übersetzt wird.

Kernighan und Ritchie, im C-Insider-Jargon meist nur noch K&R genannt, bringen als erstes Beispiel ein Programm, das den Satz »Hello, world« auf dem Bildschirm ausgibt. Da praktisch jeder C-Kurs so beginnt, wollen wir uns dem Trend nicht verschließen – aber in Deutsch.

Etwas ganz Wichtiges, das Sie bei der Eingabe der Programme beachten müssen, sollte im voraus erwähnt werden. C-Programme werden mit Vorliebe in Kleinbuchstaben geschrieben. Während die Schreibweise den meisten Basic-Interpretern völlig egal ist, unterscheidet ein Großteil der C-Compiler aber genau zwischen Namen wie »main«, »MAIN« und »Main«.

Doch nun zu unserem ersten Programm. Falls Ihnen der Ausgabetext nicht gefällt, können Sie ihn natürlich durch einen anderen ersetzen:

```
main()
{
 printf("Hallo, Welt!\n");
}
```

Sollte eine Fehlermeldung erscheinen, obwohl Sie keinen Fehler finden können, so setzen Sie vor die erste Zeile den Befehl »#include "STDIO"« oder »"STDIO.H"«.

»STDIO« ist eine Diskettendatei. Vielleicht hat sie bei Ihrem Compiler einen anderen Namen. Lassen Sie sich am besten das Inhaltsverzeichnis der Compilerdiskette ausgeben.

Wenn Sie das Programm starten, erscheint auf dem Bildschirm:

```
Hallo, Welt!
```

Sie sind natürlich nicht gezwungen, das C-Programm sklavisch nach der Vorlage einzugeben, denn der Compiler erwartet kein festes Format. Ebensogut könnten Sie schreiben:

```
main() { printf ("Hallo, Welt!
\n");}
```

Oder auch:

```
                    main() {
printf("Hallo, Welt!\n");
             }
```

Denken Sie aber an folgendes: Wenn Sie Ihre Programme nach einigen Monaten wieder anschauen, wollen Sie auch verstehen, was Sie programmiert haben. Zur logischen Gliederung eignen sich die Einrückungen hervorragend. Der Compiler versteht nämlich Programme in jedem Bildschirmformat – Sie auch?

Gehen wir der Reihe nach die Befehle durch. C-Programme bestehen ausschließlich aus »Funktionen«. Funktionen in C sind das, was man in Basic mit GOSUB oder FNx(y) aufruft und was in Pascal mit FUNCTION oder PROCEDURE bezeichnet wird. In C existiert also kein Unterschied zwischen Funktionen und Unterprogrammen.

Jedes C-Programm besteht aus mindestens einer Funktion. Die Hauptfunktion erhält den Namen »main« und wird beim Programmstart automatisch aufgerufen. »main« kann von sich aus andere Funktionen – sofern vorhanden – aufrufen. Dabei werden in Klammern Variablen als Argumente übergeben. Da das in unserem Fall nicht nötig ist, müssen Sie dem Funktionsnamen leere Klammern folgen lassen: »main()«

Der gesamte Programmcode, der hier lediglich die printf-Anweisung enthält, wird mit geschweiften Klammern ({ und }) umschlossen. Nach jedem Befehl steht ein Strichpunkt. Dieser trennt – wie in Pascal – die einzelnen Befehle voneinander.

printf hat die Aufgabe, Texte und Zahlen formatiert auszugeben. printf gehört nicht zur Sprache C, sondern ist eine Funktion. C selbst ist eine Sprache mit sehr geringem Umfang. Insbesondere fehlen Ein- und Ausgabefunktionen völlig. Die eigentlichen C-Befehle dienen vorrangig der Steuerung des Programmflusses. Die Funktion printf finden Sie hingegen bei Ihrem Compiler wahrscheinlich auf der Diskette in einer Bibliotheksdatei mit dem Namen »STDIO« oder »STDIO.H«, das steht für »Standard Input/Output«. Entsprechend der Normung der C-Sprache fallen auch überraschende Übereinstimmungen zwischen den C-Bibliotheken verschiedener Compiler auf. Sie orientieren sich fast alle an der Unix-Bibliothek von Kernighan und Ritchie.

printf gibt den String aus, der in Anführungszeichen steht. Das Zeichen »\« dient der Ausgabesteuerung. Nachfolgende Buchstaben werden zum Beispiel dazu verwendet, den nächsten Tabulatorstop zu suchen oder einen Zeilenvorschub auszulösen:

\b	Backspace (Cursor ein Zeichen nach links bewegen)
\f	Form Feed (Löschen des Bildschirms)
\n	Newline (Wagenrücklauf und Zeilenvorschub)
\r	Return (Wagenrücklauf)
\t	Tab (nächste Tabulatorposition anspringen)
\0	Nullbyte (String-Begrenzer)
\\	Das Zeichen »\« selbst
\'	Einfaches Anführungszeichen (Apostroph)
\"	Anführungszeichen (")

Statt dieser Kennbuchstaben dürfen Sie auch direkt Bytes im Oktalsystem angeben:

```
printf("\0 \1 \2 \3");
```

Die Werte zwischen oktal 0 (dezimal 0) und oktal 37 (dezimal 31) dienen bei den meisten Computern für Bildschirm-Steuerungsfunktionen, wie Farbenauswahl, Inversdarstellung, Bildschirmmodus und ähnliche Dinge. Da die Codes nicht genormt sind, können C-Programme, die auf diese Funktionen angewiesen sind, nicht ohne weiteres auf andere Computer übertragen werden.

Experimentieren Sie ruhig mit diesen Steuermöglichkeiten. Denn mit ihnen läßt sich schon einiges Interessante anstellen:

```
main()
 {
  printf("\f Zeile1\n Zeile2\t
         Tab1\t Tab2 \\");
}
```

Aber die Funktion printf kann noch viel mehr. Sie rechnet beispielsweise Zahlen in verschiedene Zahlensysteme um und druckt diese dann formatiert aus. Dazu geben Sie statt des Textes einen Format-String an, der zum Beispiel so aussieht:

```
main()
{
  printf("%x",32767);
}
```

Die Formatkenner werden durch ein Prozentzeichen eingeleitet. Danach folgt ein Buchstabe, der das Format angibt: %x besorgt beispielsweise die hexadezimale Ausgabe einer Zahl, statt 32767 wird hier also »7FFF« ausge-

druckt. Die wichtigsten Formatkenner lauten:

%d	Dezimale Ausgabe einer Integerzahl (decimal)
%o	Oktale Ausgabe einer Integerzahl (octal)
%u	Dezimale vorzeichenlose Darstellung (unsigned)
%x	Hexadezimale Ausgabe (hexadecimal)
%c	Ausgabe als ASCII-Zeichen (character)

Die Anzahl der Formatkenner muß mit der der angegebenen Zahlen übereinstimmen. Gibt es da Unterschiede, führt das meist zu recht unsinnigen Resultaten, die mitunter recht amüsant sind. C schreibt vor, daß alle Variablen, die in einem Programm benutzt werden, definiert sein müssen. Diese Definition erfolgt im Programmkopf. Die wichtigsten Datentypen in C sind:

int	Integerzahlen zwischen -32768 und 32767
char	Einzelne Buchstaben, (ASCII-Code von 0 bis 255)
float	Fließkommazahlen (Wertebereich vom Compiler abhängig)

Abwandlungen dieser Typen sind möglich. Zum Beispiel gibt es long int (32-Bit-Integerwerte – 0 bis 4294967295), short int (8-Bit-Integerzahlen - 0 bis 255), unsigned int (16-Bit-Integers ohne Vorzeichen, Wertebereich daher von 0 bis 65535) und double, das sind doppeltgenaue Fließkommazahlen. Allerdings sind die Bitbreiten und Wertebereiche nicht von Ritchie festgelegt worden. Sie werden meistens aus der Maschinensprache eines Prozessors direkt abgeleitet. So verarbeitet ein Großcomputer mit einem 32-Bit-Prozessor sicher auch »int« als 32-Bit-Integerzahlen.

Strings (Zeichenketten) sind in der C-Sprache nicht vorgesehen. Sie werden aus einzelnen Zeichen oder Feldern des Typs char zusammengestellt. Wertezuweisungen erfolgen auf die den Basic-Programmierern bekannte Art:

```
main()
{
 int x; /* Definition von x als
           Integer */
 x=64;  /* Zuweisung von 64 an
           x */
 printf("%d %x %o %c",x,x,x,x);
}
```

Kommentiert werden Programme wie oben angegeben: »/* Das ist ein Kommentar */«. Kommentare dürfen überall stehen, wo auch ein Leerzeichen oder ein Wagenrücklauf stehen könnte – nur nicht in Zeichenketten. Schachteln Sie auch keine Kommentare - das geht garantiert schief. Diese Einschränkungen sollten Sie aber nicht daran hindern, Ihre Programme möglichst ausführlich mit Kommentar-Zeilen zu erläutern. Wie bei allen Compilersprachen wirken sich Kommentare nicht auf die Länge oder die Ablaufgeschwindigkeit des erzeugten Programms aus.

»int x« bestimmt, daß die Variable x im Programm eine Integervariable ist. »x=64« weist der Variablen den Wert 64 zu. printf schließlich gibt x in verschiedenen Formaten aus: dezimal (64), hexadezimal (40), oktal (100) und als ASCII-Zeichen (@).

Statt einer Einzelvariablen können Sie auch mehrere auf einen Schlag angeben:

```
int   x,y,z;
char  c1,c2,c3,c4;
float a,b,g,h;
```

Mit den C-Variablen läßt sich genauso rechnen wie in Basic:

```
main()
{
  int x,y;
  y=23+4*5;
  x=y*(2+y-3);
  y=x/2;
  printf("%d %d",x,y);
}
```

Alle vier Grundrechenarten sind also vorhanden (Addition »+«, Subtraktion »–«, Multiplikation »*« und Division »/«). Leider ist die Hierarchie der Operatoren in C ziemlich kompliziert und auch nicht immer genau festgelegt. Wenn Sie also Zweifel an der Ablauffolge haben, setzen Sie sicherheitshalber Klammern. Das ist jederzeit erlaubt.

Da C eine sehr maschinennahe Sprache ist, entdecken Compiler und Laufzeitbibliothek (das ist der Code, der zu jedem übersetzten Programm dazugebunden wird, damit es ablaufen kann) normalerweise keine Über- oder Unterschreitungen des Wertebereichs. Die Programme schneiden einfach den Teil der Zahl weg, den sie nicht verwerten können. Wenn Sie zum Beispiel als Ergebnis einer ganzzahligen Rechnung mit Integervariablen eine 20 Bit breite Zahl erhalten, »vergißt« das Programm die vier vordersten Bits und läßt nur die sechzehn darstellbaren übrig.

Was wäre eine Programmiersprache wert, die jedes Programm nur von Anfang bis Ende abarbeiten könnte, ohne den Programmfluß zu steuern? Basic ist auf diesem Gebiet recht spartanisch ausgestattet, denn die meisten Interpreter bieten nur IF-THEN- und FOR-NEXT-Konstruktionen an – wenn es hochkommt vielleicht auch WHILE-WEND. Und natürlich das berühmt-berüchtigte, eher verwirrende GOTO.

Mit IF kann man auch in C programmieren. Nur ist die Syntax der if-Anweisung (Kleinschreibung in C!) etwas anders:

```
if (Bedingung) Aktion;
```

Trifft die Bedingung, die in Klammern gesetzt ist, zu, wird der Befehl »Aktion« ausgeführt. Setzen Sie aber keinesfalls zwischen Bedingung und Aktion einen Strichpunkt. In diesem Fall interpretiert der Compiler den Strichpunkt als Leerbefehl. Die Aktion wäre dann unabhängig von der if-Bedingung und würde immer ausgeführt werden.

Ein Programmbeispiel:

```
main()
{
 int c;
 c=getchar();
 if (c==64) printf (" Klammeraffe
                     !\n");
            printf (" Der ASCII-
            Code ist %d",c);
}
```

Die Funktion getchar() stammt aus der Bibliothek und liest ein Zeichen von der Tastatur ein. Sie übergibt der Variablen c den ASCII-Code des Zeichens. Wenn Sie den »Klammeraffen«, das ist das Zeichen »@«, eingegeben haben, meldet der Computer »Klammeraffe!«, und gibt zusätzlich dessen ASCII-Code aus.

Der Vergleich »c==64« ist keineswegs ein Druckfehler. Damit der Compiler ihn von einer Wertzuweisung »c=64« unterscheiden kann, müssen beim Vergleich zwei Gleichheitszeichen angegeben werden.

Hier alle Vergleichsoperatoren:

==	x==y	x ist gleich y
<	x<y	x ist kleiner als y
<=	x<=y	x ist kleiner oder gleich y
>	x>y	x ist größer als y
>=	x>=y	x ist größer oder gleich y
!=	x!=y	x ist ungleich y

Etwas ungewohnt dürfte die Schreibweise »!=« sein. Eine Eselsbrücke: Stellen Sie sich die beiden Symbole übereinander gedruckt vor. Dann erhalten Sie ein Zeichen ähnlich dem mathematischen für die Ungleichheit (»≠«).

Was machen Sie aber, wenn Sie nach der if-Bedingung nicht nur einen einzelnen Befehl, sondern eine ganze Befehlsgruppe ausführen lassen wollen? Sie erklären einfach die Befehle zu einem Block und umgeben diese Verbundanweisung mit geschweiften Klammern. Das entspricht exakt der BEGIN-END-Schachtelung in Pascal.

```
main()
 {
   int c;
   c=getchar();
   if (c != '*') {
              printf("Dies ist
              kein Sternchen
              !\n");
              printf("Es ist
              ein %c",c);
              }
}
```

Wenn Sie ein Zeichen eingeben, das kein Stern ist (ASCII-Code 42), werden die beiden printf-Funktionen ausgeführt. Andernfalls beendet der Computer das Programm. In der if-Zeile sehen Sie, daß auch String-Konstanten als Vergleichsobjekte dienen können:

```
if (c != 42)
if (c != '*')
if (c != 0x2A)
```

sind alle identisch.

Hexadezimale Zahlen müssen mit »0x« beginnen – zum Beispiel »0xFF«, »0x3C« oder »0x3FAC«.

Die if-Anweisung läßt sich um einen else-Teil erweitern, der ausgeführt wird, wenn die gegebene Bedingung nicht zutrifft:

```
main()
 {
   int c;
   c=getchar();
   if (c != 0x2A) printf
                 ("Kein Stern!");
                  else printf
                 ("Ein Stern!");
 }
```

Auch hier ist eine Verbundanweisung zulässig, die die Ausführung mehrerer von if oder else abhängiger Befehle erlaubt:

```
main()
 {
  int c;
  c=getchar();
  if (c != 42) { printf("Kein
                 Stern!$\n");
                 printf("Statt
                 dessen ein %c",
                 c);
               }
          else { printf("Ein
                 Stern!$\n");
                 printf("Der
                 ASCII-Code ist
                 42");
               }
  }
```

Wenn Sie eine Vorliebe für verwirrende Programme haben, können Sie auch if-Kommandos ineinander verschachteln. Wie wäre es mit folgendem Programm?

```
main()
 {
   int c;
   c=getchar();
   if (c >= 42) { if (c == 42)
         printf("ASCII-Code=42");
    else printf("ASCII-Code>42");
                 }
    else printf("ASCII-Code<42");
 }
```

Alles klar? Gerade dieses Programm läßt sich erheblich verständlicher mit drei gleichrangigen if's schreiben:

```
main()
 {
   int c;
   c=getchar();
   if (c>42) printf
             ("ASCII-Code>42);
   if (c==42) printf
             ("ASCII-Code=42);
   if (c<42) printf
             ("ASCII-Code<42);
 }
```

Für den Fall, daß ein Befehl oder eine Befehlsgruppe von mehreren Bedingungen gleichzeitig abhängt, könnten Sie – wie oben gezeigt – mehrere if's zusammenbasteln. Wesentlich übersichtlicher wird es aber mit logischen Operatoren, die den bekannten Funktionen AND, OR und NOT aus Basic entsprechen:

&& Logisches Und (AND)
|| Logisches Oder (OR)
! Logisches Nicht (NOT)

Demonstrieren lassen sich die Operatoren mit einem Programm, das drei Zeichen von der Tastatur entgegennimmt und dann vergleicht:

```
main()
 {
  int x,y,z; /* Speichern die
                3 Zeichen */
  x=getchar(); getchar();
  y=getchar(); getchar();
  z=getchar(); getchar();
  if (x=='A' && y=='B' && z=='C')
        printf("Alphabet!\n");
        /* 1 */
  if (x>='1' && x<='9')
        printf("x ist Ziffer\n");
        /* 2 */
  if (x=='+' || y=='+')
        printf("x oder y = +");
        /* 3 */
 }
```

Im Fall /*1*/ gibt der Computer den Text »Alphabet!« aus, wenn Sie A, B und C eingeben. Fall /*2*/ meldet »Ziffer!«, wenn das erste Zeichen Ihres Inputs (entspricht der Variablen x) eine Ziffer ist. Der Fall /*3*/ zeigt das logische Oder. Ist das erste oder das zweite Zeichen ein Plus »+«, schreibt das Programm die Meldung »x oder y = +« auf den Bildschirm.

Die Gliederung

```
x=getchar(); getchar();
y=getchar(); getchar();
z=getchar(); getchar();
```

ist notwendig, weil die getchar-Funktion auch den Code der Return- oder Enter-Taste (10 oder 13) registriert. Um diesen jeweils auszufiltern, muß ein »Dummy«-getchar eingesetzt werden. An diesem Beispiel ist recht gut zu erkennen, daß C-Funktionen entweder keinen oder genau einen Parameter ans aufrufende Programm übergeben können. x=getchar() fragt die Tastatur ab und liefert den Code der gedrückten Taste in der Variablen x ab. getchar() macht das gleiche, doch der Tastenwert geht verloren. Eine solche Funktion, die keinen Wert zurückgibt, ist eigentlich nichts anderes als ein Unterprogramm oder eine Prozedur in einer der herkömmlichen Programmiersprachen.

Neben diesen logischen Operatoren befinden sich noch zahlreiche weitere im C-Sprachumfang:

– Der Minus-Operator: -a=0-a. Dieser Operator zeigt an, daß eine Zahl negativ ist, genau wie in Basic (beispielsweise »B=-A« oder »Z=-3*A«). Seltsamerweise ist in C kein Plus vor Variablen und Zahlen erlaubt. x=+a ist also verboten – aber eigentlich auch unsinnig. Hingegen ist x=b+a eine Selbstverständlichkeit. So können Sie statt x=+a auch x=0+a schreiben. Allzu viel Sinn steckt aber nicht hinter solch einer Befehlsfolge.

– Das Einerkomplement: ~a dreht alle Bits in der angegebenen Variablen oder Konstanten um. Aus Nullen werden Einsen und aus Einsen Nullen. Damit ist ~255=-256.

– Die Inkrementoperatoren: a++ und ++a. Beide Operatoren erhöhen den Wert von a um Eins. Der Unterschied liegt im Zeitpunkt, zu dem das geschieht:

`a=3; printf("%d",++a);`

druckt "4", a ist 4.

++a ist ein Präfix-Operator.

`a=3; printf("%d",a++);`

druckt "3", a ist 4.

a++ ist ein Postfix-Operator.

Mit den Inkrement-Anweisungen ist eine Zählschleife sehr leicht zu konstruieren:

```
main()
 {
   int i;
   i=0;
   while (i>1000) printf
                 ("%d\n",i++);
 }
```

Ausführlich gehen wir auf while weiter unten ein.

– Die Dekrementoperatoren: a-- und --a.

Sie vermindern den Wert von a um Eins. Es gelten dieselben Regeln wie bei den Inkrementbefehlen.

– Die Shift-Befehle: Sie schieben die Bits in einer Variablen um einen bestimmten Wert nach links oder rechts.

`a=2; a<<1`

ist der Links-Shift. a hat den Wert 4 (2*2).

`a=8; a>>2`

ist der Rechts-Shift. a hat den Wert 2.

– Die bitweisen Logikoperatoren: Die oben erklärten Logikoperatoren && und \\ liefern immer nur die Werte 0 und 1, je nachdem, wie die beiden Operanden aussehen. Die bitweisen Logikoperatoren verknüpfen die Operanden Bit für Bit und weisen dann der angegebenen Variablen das Ergebnis zu.

a&b ist das bitweise logische Und. 127 & 255 ergibt 127 (0111 1111 bin AND 1111 1111 bin = 0111 1111 bin).

a\b ist das bitweise logische Oder. 128 \ 3 ist 131 (1000 0000 bin OR 0000 0011 bin = 1000 0011 bin).

a↑b hat in C absolut nichts mit der Potenzfunktion zu tun, sondern steht für das bitweise logische Exklusiv-Oder. 1 ↑ 1 ist 0 (01 bin XOR 01 bin = 00 bin).

(Martin Kotulla/hg)

Erste Schleife, zweite Schleife – C

Wiederholung – das ist eins der Zauberworte der Computer. C kennt natürlich auch Schleifen und andere Programmstrukturen, die bestimmte Befehlsfolgen immer wieder aufrufen.

C kennt verschiedene Wege, Schleifen zu konstruieren. Um sie alle zu zeigen, geben wir in unseren Beispielsprogrammen den gesamten ASCII-Zeichensatz auf dem Bildschirm aus – und zwar in folgender Form:

DEZ	HEX	OKT	ASCII
32	20	40	
33	21	41	!
34	22	42	"
.	.	.	
.	.	.	
.	.	.	
255	FF	377	

In Basic wäre wohl die einfachste Art, so etwas zu programmieren, die:

```
100 PRINT "DEZ HEX OKT ASCII"
110 FOR I=32 TO 255
120 PRINT I;HEX$(I);OCT$(I);
    CHR$(I)
130 NEXT I
```

Voraussetzung ist natürlich, daß der Basic-Interpreter die Funktionen HEX$ und OCT$ versteht. HEX$ ist ja noch relativ geläufig, aber Oktalzahlen ?

C bietet die Zahlenkonversionen grundsätzlich mit der printf-Funktion. Schwierigkeiten bereitet uns aber noch die Übersetzung der FOR-Schleife in C. »for« ist gleichzeitig auch der Befehlsname in C. Nur der Aufbau ist anders als in Basic:

```
for (Ausgangsbedingung;
    Endbedingung; Zählbedingung)
```

Eine Schleife, die von 0 bis 10000 zählt und diese Zahlen ausgibt, gibt man in C so ein:

```
for (i=0; i<=10000; i=i+1)
    printf("%d ",i);
```

Viel schwerer als in Basic gestaltet sich die Ausgabe des Zeichensatzes auch nicht.

Das Programm sieht so aus:

```
main()
{
    int i;
    printf("DEZ HEX OKT
          ASCII\n\n');
    for (i=32; i<=255; i=i+1)
        printf("%d %x %o %c
                \n",i,i,i,i);
}
```

i=32 ist die Startbedingung. Der Variablen i wird hier der Wert 32 zugewiesen. Dann führt der Computer den printf-Befehl aus. Er kehrt zur for-Anweisung zurück und prüft, ob die zweite Bedingung i<=255 noch zutrifft. Ist sie nicht mehr erfüllt, beendet das Programm die Schleife. Sonst gilt i=i+1, und die Schleife wird erneut abgearbeitet.

Der besondere Aufbau der for-Schleife schreibt nicht zwingend vor, bei allen drei Bedingungsteilen dieselbe Variable zu verwenden. Sie können sogar einzelne Teile völlig weglassen. Ein Programm, das eine Zeichenkette von der Tastatur liest und in der Folge auf dem Bildschirm ausgibt, kann damit so aussehen:

```
main()
{
    int c;
    c=0;
    for ( ; c != 10 ; )
        {c=getchar();
            printf("%c",c);
        }
}
```

Die Schleife druckt solange Zeichen aus, bis sie ein Line-Feed (ASCII-Code 10) entdeckt. Sie ist übrigens ein hervorragendes Beispiel dafür, daß ein Programm – trotz nahezu vollständiger Kompatibilität der Sprache – auf unterschiedlichen Computern anders reagiert. Manche Computer puffern die Eingabe, das heißt, sie legen die gelesenen Zeichen erst intern im Speicher ab, bis der Benutzer die Zeilenschaltung (ENTER oder RETURN) betätigt. Andere Geräte verwenden die unmittelbare Ein- und Ausgabe. Je nachdem, welchen Computer und welches Betriebssystem Sie benutzen, erhalten Sie für den Satz »Dies ist ein C-Text«:

Dies ist ein C-Text Dies ist ein C-Text

oder

DDiieess iisstt eeiinn CC-TTeexxtt

In vielen Fällen ist aber eine for-Schleife nicht der Weisheit letzter Schluß. Deshalb bietet C zwei weitere Konstruktionen an: »while« und »do-while«. Beginnen wir mit »while«. Dieser Befehl führt die nachfolgenden Anweisungen oder die folgende Befehlsgruppe aus, solange eine Bedingung erfüllt ist:

```
while (Bedingung) Aktion;
```

Das Zeichensatz-Programm läßt sich mit while umschreiben – auch wenn es dadurch unübersichtlicher wird:

```
main()
{
    int i;
    i=32;
    while (i<=255)
      {
        putchar(i);
        i=i+1;
            }
}
```

Damit der Compiler weiß, was alles abhängig von while ausgeführt werden soll, müssen Sie die Befehle zu einer Verbundanweisung zusammenfassen und mit »{« und »}« umklammern.

Neu ist hier die Funktion putchar. Sie gibt ein einzelnes Zeichen vom Typ Integer oder Char auf dem Bildschirm aus. Gleichwertig dazu steht »printf("%c", i)«. printf erzeugt zwar einen längeren Programmcode, dafür ist diese Funktion aber auch universeller. Allerdings vermag putchar durchaus auch \-Formatkenner (\f zum Löschen des Bildschirms, \t für Tabulatorsprung und so weiter) zu verarbeiten. Ein putchar('\f') ist – von der Programmlänge her gesehen – einem printf("\f") vorzuziehen.

while prüft vor der Ausführung der Schleife, ob die angegebene Bedingung zutrifft. Das kann auch bedeuten, daß die Schleife unter Umständen überhaupt nicht bearbeitet wird:

```
main()
{
    while (3>4) putchar(64);
}
```

Da 3 natürlich niemals größer als 4 ist, wird kein einziger Klammeraffe (ASCII-Code 64) ausgegeben.

Anders arbeitet do-while:

```
do
   Aktion;
 while (Bedingung);
```

Hier prüft der Computer erst nach mindestens einmaliger Ausführung der Aktion, ob die Bedingung wahr ist.

```
main()
  {
    do
      putchar(64);
    while (3>4);
  }
```

Natürlich benötigt do-while keine Klammerung der abhängigen Anweisungen, da diese do und while schon umschließen. So sind die Befehle für den Compiler als Verbundanweisung erkennbar, und es können gar keine Verständniskonflikte auftreten.

Manchmal tritt der Fall ein, daß der Computer eine Schleife vor dem eigentlichen Ende beenden soll, zum Beispiel wenn ein Ergebnis einer Berechnung bereits vorliegt oder ein bevorstehender Fehler abgefangen werden soll. Dafür kennt C den break-Befehl. Ihn demonstriert ein Programm, das eine Eingabe von der Tastatur entgegennimmt und die ASCII-Codes aller Zeichen addiert. Abgebrochen werden soll das Programm, wenn ein Sternchen (ASCII-Zeichen 42) entdeckt wird:

```
main()
  {
    int c,summe;
    summe=0; /* Mit Anfangswert
                initialisieren */
    while (1) /* Endlosschleife */
     {
       c=getchar();
       if (c==42) break;
       summe=summe+c;
     }
    printf("Summe: %d",summe);
  }
```

Das Programm definiert sich die zwei Variablen c und summe. c empfängt jeweils ein Zeichen von der Tastatur, und summe addiert die ASCII-Codes. »while(1)« ist ein beliebter Trick, um den Computer in einer Endlosschleife festzuhalten. Da die Bedingung immer erfüllt ist, beendet das Programm die Schleife nie aufgrund dieser while-Anweisung. Also muß irgendeine andere Abbruchbedingung gefunden werden, in unserem Beispiel »if (c==42) break;«. Nachdem der Computer ein Zeichen von der Tastatur gelesen hat (c=getchar();), prüft er nach diesem Befehl, ob es sich um ein Sternchen handelt. break befiehlt dem Computer, die while-Schleife zu verlassen und als nächstes den ersten Befehl hinter der Schleife abzuarbeiten. Das ist in diesem Programm eine printf-Funktion, die die Summe der ASCII-Codes ausgibt.

Ebenso läßt sich break auch in for- und do-while-Schleifen verwenden. Auch dort bewirkt der Befehl einen Abbruch der Schleifenausführung:

```
main()
  {
    int i;
    for (i=0; i<1000; i=i+1)
      {
        printf("%d\n",i);
        if (i==500) break;
      }
    printf("Ende!");
  }
```

Die for-Anweisung schreibt dem Computer zwar vor, von 0 bis 1000 zu zählen und die einzelnen Werte auszugeben, doch er bricht aufgrund der Zeile »if (i==500) break« die Zählung bei 500 ab.

Den gegenteiligen Effekt von break bewirkt continue: continue ruft diejenige Programmzeile auf, in der die Schleifenbedingung getestet wird, und führt damit die restlichen Kommandos, die eventuell noch in der Schleife folgen, nicht aus. Der Befehl bezieht sich immer auf die innerste Schleife. Wenn mehrere Schleifen ineinander geschachtelt sind, kann continue also nicht benutzt werden, um eine äußere Schleife aufzurufen.

Ein Beispiel für continue in einer for-Schleife:

```
main()
 {
   int i;
   for (i=0; i<100; i=i+1)
    {
      if ((i % 10) != 0) continue;
      printf("%d\n",i);
    }
 }
```

Dieses C-Programm gibt nacheinander die Zahlen 10, 20, 30, 40 und so weiter bis 90 aus: es zählt in der for-Schleife von 1 bis 100 in Einerschritten hoch. In der if-Zeile werden aber alle Zahlen, die keine Zehner sind, ausgefiltert. Dazu bietet C einen Modulo-Operator, das Prozentzeichen. Dieser Rechenoperator gibt den Rest einer ganzzahligen Division an:

```
20/3=6,    6*3=18,    20-18=2    Rest:  20 % 3 = 2
40/5=8,    8*5=40,    40-40=0    Rest:  40 % 5 = 0
191/3=63,  63*3=189,  191-189=2  Rest: 191 % 3 = 2
```

In Pascal und in manchen Basic-Dialekten heißt dieser Operator »mod«:

```
100 INPUT a,b
110 PRINT a\b,a MOD b
120 GOTO 100
```

Läßt sich eine Zahl nicht ohne Restbetrag durch 10 teilen, ist sie keine Zehnerzahl. Sie wird folglich von continue »verschluckt«. Die übrigen Zahlen gibt die printf-Funktion auf dem Bildschirm aus.

Neben if gibt es in C noch eine weitere Möglichkeit, Vergleiche anzustellen. Dieser Befehl heißt »switch-case« und ist immer dann zu empfehlen, wenn viele Vergleiche mit einer einzigen Variablen durchgeführt werden sollen. Ein Programm, das ein Symbol von der Tastatur einliest und dann untersucht, schreiben Sie so:

```
main()
 {
   int c;
   c=getchar();
   if (c==32) printf
      ("Ein Leerzeichen!\n");
   if (c==33) printf
      ("Ein Ausrufezeichen!\n");
   if (c==34) printf
      ("Ein Anführungs-
      zeichen!\n");
   if (c==35) printf
      ("Ein Doppelkreuz!\n");
   if (c==36) printf
      ("Ein Dollarsymbol!\n");
   if (c==37) printf
      ("Ein Prozentzeichen!\n");
 }
```

Spätestens nach dem zehnten if mag Ihnen das alles zu umständlich vorkommen. Es ist auch nicht die eleganteste Art zu programmieren. So geht es einfacher:

```
main()
  {
  int c;
  c=getchar();
  switch(c)
  {
  /*1*/ case 32: printf
        ("Ein Leerzeichen!\n");
  /*2*/ case 33: printf
        ("Ein Ausrufezeichen!\");
  /*3*/ case 34: printf
        ("Ein Anführungs-
        zeichen!\");
  /*4*/ case 35: printf
        ("Ein Doppelkreuz!\");
  /*5*/ case 36: printf
        ("Ein Dollarsymbol!\");
  /*6*/ case 37: printf
        ("Ein Prozentzeichen!\");
      }
}
```

Bis auf den switch-Teil mit der Verbundanweisung ist das Programm identisch mit dem vorigen. In switch wird die Variable angegeben, die zu überprüfen ist, hier die Variable c. In einer, durch geschweifte Klammern umgebenen, Verbundanweisung stehen nach case die Fälle, von denen ausgehend Befehle ausgeführt werden. Die »/*Kommentare*/« sind nur angegeben, um einen Bezug auf die Zeilen herzustellen. Die Arbeit des Computers veranschaulichen Sie sich so:

```
int c;      Ich definiere eine Variable c als Integer
switch(c)   Aha, jetzt soll ich c untersuchen
case 32:    Hat c den Wert 32?
            Ja - dann muß ich »Ein Leerzeichen!« drucken
            Nein - dann unternehme ich nichts
case 33:    Hat c den Wert 33?
            Ja - ich gebe »Ein Ausrufezeichen!« aus
            Nein - ich mache gar nichts
...
case 37:    Ist c=37?
            Ja - ich melde »Ein Prozentzeichen!«
            Nein - ich habe ja überhaupt nichts zu tun!
```

Wenn Sie das Programm mit Ihrem C-Compiler übersetzen und laufen lassen, geben Sie zuerst einmal das Prozentzeichen (%) ein. Der Computer meldet: "Ein Prozentzeichen!"

Das Programm arbeitet aber noch nicht korrekt. Denn tippen Sie statt des Prozentsymbols ein Ausrufezeichen ein, so gibt es einiges an Text aus:

```
Ein Ausrufezeichen!
Ein Anführungszeichen!
Ein Doppelkreuz!
Ein Dollarsymbol!
Ein Prozentzeichen!
```

Die Lösung für diesen »Fehler« ist recht simpel: Der case-Befehl versteht sich nämlich nicht wie Pascals CASE, sondern eher wie der GOTO-Befehl in diesem Basic-Programm:

```
10 INPUT a$
20 ON ASC(a$)-32 GOTO 32,33,34,35,
    36,37
32 PRINT "Ein Leerzeichen!"
33 PRINT "Ein Ausrufezeichen!"
34 PRINT "Ein Anführungszeichen!"
35 PRINT "Ein Doppelkreuz!"
36 PRINT "Ein Dollarsymbol!"
37 PRINT "Ein Prozentzeichen!"
```

case durchläuft also alle folgenden Fälle und führt sie ohne Prüfung der Bedingung aus. Da dies meistens unerwünscht ist, ist Abhilfe bereits vorgesehen: Der break-Befehl, den wir vorhin im Zusammenhang mit Schleifenkonstruktionen erwähnt haben, verläßt ebenso eine switch-case-Anweisung:

```
main()
{
  int c;
  printf("Bitte die Nummer
          eines Wochentags: ");
  c=getchar()-48;
  switch(c)
   {
     case 1: printf
             ("Montag"); break;
     case 2: printf
             ("Dienstag"); break;
     case 3: printf
             ("Mittwoch"); break;
     case 4: printf
             ("Donnerstag");
             break;
     case 5: printf
             ("Freitag"); break;
     case 6: printf
             ("Samstag"); break;
     case 7: printf
             ("Sonntag"); break;
   }
  printf(" ist der %d. Wochen-
         tag.",c);
}
```

Wenn Sie hier eine Zahl zwischen 1 und 7 eingeben, antwortet der Computer mit dem korrespondierenden Wochentag:

```
Bitte die Nummer eines Wochentags: 1
Montag ist der 1. Wochentag.
Bitte die Nummer eines Wochentags: 4
Donnerstag ist der 4. Wochentag.
Bitte die Nummer eines Wochentags: 7
Sonntag ist der 7. Wochentag.
```

Entfernen Sie übungsweise einmal überall die break-Anweisungen. Sie sehen dann, wie der Computer alle Wochentage »durchrutschen« läßt.

Falls Sie ein Symbol eingegeben haben, das keine Ziffer oder aber die Zahlen »8« oder »9« darstellt, ignoriert der Computer einfach die case-Anweisungen und macht nach der Verbundanweisung – hier also bei »printf« – weiter, als wäre nichts geschehen. Oft ist aber eine Fehlermeldung angebracht. Der Befehl, der es erleichtert, solche Programm- oder Eingabefehler abzufangen, heißt »default«:

```
main()
{
  int i;
  printf("Wählen Sie:A oder B");
  i=getchar();
  switch(i)
   {
     case 'A':  printf
     ("Ein A also!");
     break;
     case 'B':  printf
     ("Sie haben B getippt!");
     break;
     default:   printf
     ("Eingabefehler! ");
     printf
     ("Weder A noch B.");
   }
}
```

Das Programm fordert von der Tastatur einen Buchstaben an, wahlweise »A« oder »B«. Für diese beiden Symbole ist das Verhalten genau definiert. Der Computer führt die zugehörige case-Anweisung aus. Bei anderen Eingaben ruft er »default« auf.

Auch die Ausführung eines einzelnen Befehls auf mehrere case-Anweisungen hin ist zulässig:

```
switch(x)
case 1:
case 2: printf("1 oder 2");
```

Das »Wochentags-Programm« kann man so zu einem »Wochenende-Programm« machen:

```
main()
 {
   int c;
   printf("Bitte die Nummer
           des Wochentags: ");
   c=getchar()-48;
   switch(c)
    {
      case 1:
      case 2:
      case 3:
      case 4:
      case 5: printf
      ("Wochentag"); break;
      case 6:
      case 7: printf
      ("Endlich Wochenende!");
      break;
    }
 }
```

Die Fälle 1 bis 5 werden mit Wochentag beantwortet. Entdeckt der Computer aber eine 6 oder 7, meldet er freudig »Endlich Wochenende!«

Man glaubt es kaum, welchen Horror ein kleines Wörtchen auf viele Programmierer ausübt. Struktur-Fanatiker scheuen den Befehl »GOTO« in etwa demselben Maße wie Vampire den heranbrechenden Morgen. Es stimmt sicher, daß die allzu häufige Anwendung von GOTO nicht gerade zu übersichtlichen Programmen führt. Aber in manchen Fällen ist dieser unbedingte Sprung in allen Programmiersprachen sehr nützlich. Letztlich gibt es GOTO sogar in vielen Pascal-Versionen. C steht da nicht zurück:

```
goto sprungziel;
 ...
 sprungziel: ...
```

Das folgende Beispiel eignet sich bestens, um zu zeigen, daß sich GOTO aber fast immer vermeiden läßt:

```
main()
   {
    int i;
    i=32;
    sprungziel: printf("%d = %x
                = %o = %c{n"
                ,i,i,i,i);
                i=i+1;
                if (i<=255)
                goto sprungziel;
   }
```

Das Programm gibt, wie unschwer zu erkennen ist, den ASCII-Zeichensatz und die verschiedenen Zahlenäquivalente in dezimaler, hexadezimaler und oktaler Notation aus. Erinnern Sie sich noch, wie elegant im Gegensatz dazu die Verwendung der for-Schleife wirkt?

Bisher verwendeten Sie nur die Funktion main(), die das eigentliche Hauptprogramm darstellt. Ebenso einfach können Sie Unterfunktionen definieren. Ob Sie diese vor oder hinter main() ablegen, ist dem Compiler völlig egal.

```
main()
   {
    printf
    ("Ich bin in main()\n");
    subfunction(); /* Aufruf
    der Funktion */
    printf
    ("Ich bin wieder main()\n");
   }

 subfunction()
   {
    printf
    ("Ich bin subfunction()\n");
   }
```

Genauso gut können Sie subfunction() vor main() angeben.

```
subfunction()
   {
    printf
    ("Ich bin subfunction()\n");
   }
main()
   {
    printf
    ("Ich bin main()\n");
    subfunction(); /* Aufruf der
    Funktion */
    printf("Ich bin
    wieder main()\n");
   }
```

Beim Programmstart wird main() aufgerufen und gibt den Text »Ich bin in main()« aus. Dann findet der Computer



den Aufruf der Unterfunktion subfunction() und bearbeitet diese. Dort steht der Ausgabebefehl »Ich bin subfunction()«. Da diese Funktion gleich wieder zu Ende ist, kehrt das C-Programm nach main() zurück. Es führt den nachfolgenden printf-Befehl aus: »Ich bin wieder main()«.

Die Bildschirmausgabe sieht also insgesamt so aus:

```
Ich bin main()
Ich bin subfunction()
Ich bin wieder main()
```

Am Ende des Funktionskopfes, also dort, wo der Namen der Funktion steht, dürfen Sie keinesfalls einen Strichpunkt setzen. Denn der Compiler verstünde dies als Funktionsaufruf anstatt als Definition.

Funktionsaufrufe lassen sich beliebig verschachteln, wenn Sie es nicht übertreiben und durch mehrere hundert Funktionen den gesamten Computer blockieren. Somit sind auch rekursive Programmiertechniken erlaubt.

Funktionen in C können aber nicht ihre »privaten« Unterfunktionen haben, wie das in Pascal erlaubt ist. In C sind alle Funktionen gleichwertig. Es gibt also keine von einer anderen Funktion abhängige Funktion, die nicht auch von einer dritten, unabhängigen Funktion aufgerufen werden kann. Diese Beschränkung gegenüber Pascal trägt sehr zur Verständlichkeit und Lesbarkeit von C-Programmen bei.

Sie dürfen also nicht so programmieren:

```
Funktionsdefinition 1
Funktionsdefinition 2
        Funktionsdefinition 2a
        Funktionsdefinition 2b
             Funktionsdefinition 2ba
             Funktionsdefinition 2bb
Funktionsdefinition 3
```

Statt dessen müssen Sie in C schreiben:

```
Funktionsdefinition 1
Funktionsdefinition 2
Funktionsdefinition 2a
Funktionsdefinition 2b
Funktionsdefinition 2ba
Funktionsdefinition 2bb
Funktionsdefinition 3
```

Während der erste Aufbau (Pascal) den Aufruf von 2bb aus 3 verbietet, ist dieser beim zweiten Schema, das den Syntax-Vereinbarungen von C entspricht, erlaubt.

Die Namen der Funktionen müssen – übrigens ebenso wie die Variablennamen – gewissen Konventionen entsprechen. So muß das erste Zeichen ein Buchstabe zwischen a und z beziehungsweise A und Z oder ein Unterstrich (_) sein. Die restlichen Zeichen sind Buchstaben, Ziffern oder Unterstriche. Von der Verwendung des Unterstrichs als erstem Symbol eines Namens ist aber abzuraten. Nach internationaler Gepflogenheit unter den C-Programmierern sind diese Namen nämlich für die interne Verwaltung des Compilers, zum Beispiel in den mitgelieferten Bibliotheksdateien, reserviert. Ein Verstoß gegen diese Regel kann Namenskollisionen zur Folge haben. Die maximal erlaubte Namenslänge hängt vom Compiler ab und wurde im C-Standard von Kernighan und Richie nicht exakt festgelegt. Bei den meisten Compilern dürfen Namen beliebig lang sein, es werden aber nur die ersten sechs oder acht Stellen unterschieden. Bei solchen Compilern sind damit die Namen »ein__langer__funktionsname1()« und »ein__langer__funktionsname2()« identisch – also aufgepaßt! Natürlich dürfen Namen einer Funktion oder Variablen nicht dem Namen eines Befehls (for, while, case und so weiter) entsprechen.

Damit sollte Ihnen der einfache Unterprogrammaufruf ohne Datenübergabe klar sein. Wichtig ist nur noch, anzumerken, daß jede Funktion – so auch main() – lokale Variablen verwendet. »Lokal« bedeutet, daß die Variablen einzig und allein für diese Funktion definiert sind. Andere Funktionen können auf sie nicht zugreifen:

```
main()
  {
    int x;
    x=3;
    printf("x ist in main()
           %d\n",x);
    subfunct();
    printf("x ist in main()
           immer noch %d\n",x);
  }
subfunct()
  {
    int x;
    x=10;
    printf("x ist in subfunct()
           %d\n",x);
    x=x*2;
    printf("x ist jetzt in
           subfunct() %d\n",x);
  }
```

Die Variable x, die in main() definiert wurde, hat überhaupt nichts mit der Variablen x in subfunct() zu tun. Ihre einzige Übereinstimmung bleibt der Name. Keine Manipulation des Wertes in main() hat irgendwelche Auswirkungen auf den Wert in subfunct(). Dementsprechend liefert das Programm die folgenden Ausgaben:

```
main(): x=3
Meldung: »x ist in main() 3«
subfunct(): x=10
Meldung: »x ist in subfunct() 10«
subfunct(): x=x*2
Meldung: »x ist in subfunct() 20«
main():
Meldung: »x ist in main() immer noch 3«
```

Sie können aber auch sogenannte globale Variablen vereinbaren. Im Gegensatz zu den lokalen Variablen, deren Wirkungsbereich sich auf eine Funktion beschränkt, gelten die globalen in allen Funktionen. Sie werden vom Compiler als allgemeingültig erkannt, indem sie an den Anfang des Programms außerhalb der Funktionen aufgeführt werden:

```
int i;
main()
  {
    i=999; /* Wertzuweisung
           in main() */
    printf("In main() hat i
           den Wert %d\n",i);
    subfunct();
  {
 subfunct()
  {
    printf("Auch in subfunct()
           ist i=%d\n",i);
  }
```

Die Variable i steht in der ersten Zeile, also noch bevor main() oder subfunct() beginnen. Somit ist i global. In main() wird der Variablen der Wert 999 zugewiesen. Daß die Zuweisung korrekt ablief, beweist die printf-Zeile: »In main() hat i den Wert 999«. Der Befehl »subfunct();« überträgt die Kontrolle an die gleichnamige Funktion. Der Aufruf von printf bezeugt, daß i den Wert 999 behalten hat. Jede Veränderung des Werts von i in einer Funktion hat die Wertveränderung in allen anderen Funktionen zur Folge – ganz einfach, weil es sich immer um dieselbe Variable handelt.

In C haben lokale Variablen eine Vorrangstellung vor den globalen. Wenn Sie eine globale Variable definiert haben, können Sie ohne Probleme eine lokale Variable gleichen Namens initialisieren. Während der Laufzeit der zugehörigen Funktion wird die lokale Variable verwendet. Nach Ende der Bearbeitung benutzt der Computer wieder die globale Variable:

```
int i=123; /* globales i
           ist 123 */
main()
  {
    printf("Globales i ist
         %d\n",i);
    subfunct();
    printf("Globales i ist
         immer noch %d\n",i);
  }
subfunct()
  {
    int i; /* lokale Definition
           von i */
    i=999; /* lokale Wertzu-
           weisung */
    printf("Lokales i ist
         %d\n",i);
  }
```

Das C-Programm meldet:

```
Globales i ist 123.
Lokales i ist 999.
Globales i ist immer noch 123.
```

(Martin Kotulla/hg)



Daten werden unter C in verschiedenen Klassen gespeichert. Ob die Variable automatisch gelöscht wird oder im Speicher stehenbleibt – das muß der Programmierer beachten.

Speicherklassengesellschaft in C

C besitzt storage-classes, was in der deutschen Übersetzung »Speicherklassen« heißt. Dieser Ausdruck stammt aus der »C-Bibel« von Kernigham und Ritchie und wird hier näher beschrieben.

Es gibt drei Speicherklassen in C, nämlich »static«, »auto« und »register«. Normale Variablen, die bei ihrer Definition nicht anders bezeichnet werden, zählen zum »automatic«-Typ. Das heißt, sobald die sie betreffende Funktion aufgerufen ist, werden die Variablen im Speicher angelegt. Beim Beenden der Funktion löscht der Computer ihre Werte und gibt den benötigten Speicherplatz wieder frei. Dadurch gehen ihre Inhalte verloren und können beim nächsten Aufruf derselben Funktion also nicht weiterbenutzt werden.

automatic-Variablen werden durch die Angabe des Schlüsselwortes »auto« spezifiziert:

```
auto int a;
auto char b;
auto float xyz;
```

Wenn Sie keine anderweitige Angabe hinzusetzen, verarbeitet er die Variablen selbsttätig als »auto«. Statt der gezeigten Definitionen können Sie also auch schreiben:

```
int a;
char b;
float xyz;
```

Diese Notation sind Sie ja bisher schon gewohnt. Ein anderes Verhalten zeigen die static-Variablen. Einmal im Speicher angelegt, sind sie dann bis zum Programmende bei jedem Aufruf der Funktion, für den die Definition gilt, verfügbar. Ihr Wert bleibt also auch beim Beenden der Funktion erhalten. Zur Verdeutlichung hier ein Programmbeispiel, das das unterschiedliche Verhalten von auto- und static-Variablen zeigt:

```
main()
{
 sub(); /* 1.Unterprogrammauf-
          ruf */
 sub(); /* 2.Unterprogrammauf-
          ruf */
}
sub();
{
     auto int x=3;
     printf("Wert von x=%d ",x);
     x=x+1;
}
```

Die Bildschirmausgabe ist »3 3«. Bei jedem Aufruf wird x neu angelegt. Ersetzen Sie das Schlüsselwort auto durch static und schauen Sie, was passiert: »3 4«. Beim ersten Aufruf von sub() wird x der Wert 3 zugewiesen und auch auf dem Bildschirm ausgegeben. Danach erhöht der Befehl x=x+1 den Wert auf 4, und der Computer beendet die sub-Funktion. Beim zweiten Aufruf von x »weiß« der Computer noch, daß x den Wert 4 hat und gibt ihn auch aus. static-Variablen eignen sich also dazu, wichtige Daten bis zum nächsten Aufruf der Funktion zu »konservieren«.

Auf Grund dieser Eigenschaft müssen Sie in jedem Einzelfall entscheiden, ob Sie nun auto oder static vorziehen. Auto-Variablen haben den Vorteil, daß Speicherplatz nur für wirklich benötigte Variablen benutzt und dieser auch baldmöglichst wieder freigegeben wird. Die zugehörigen Mechanismen der Variablenverwaltung kosten allerdings sowohl Zeit als auch Programmspeicherplatz. statics erlauben damit kürzere und schnellere Programme.

Die Speicherklasse »register« weist den Compiler an, eine Variable direkt in einem der Register des Mikroprozessors abzulegen. Der Prozessor kann sie so durch einen einzigen Maschinenbefehl sehr schnell erreichen, was sich entsprechend positiv auf die Arbeitsgeschwindigkeit des Programms auswirkt. Die »herkömmlichen« Prozessoren wie Z80, 6502, 6809 und sogar der 68000 besitzen aber gar nicht genügend Register, um noch einige zusätzlich einem C-Programm zur Verfügung zu stellen. Die diversen C-Compiler für kleinere Computer geben deshalb bei diesem Befehl entweder eine Fehlermeldung aus (sehr selten), oder sie ignorieren die Anweisung und machen aus »register« »auto« (das ist bei Compiler-Autoren weitaus beliebter). Sie werden nun sicher fragen: Wenn ich jetzt schon Funktionen definieren kann, muß ich doch irgendwie zwischen den einzelnen Funktionen Daten austauschen können. Bloß wie?

Basic-Programmierer werden antworten: Kein Problem. Wir haben doch die, ach so praktischen, globalen Variablen. Nun definieren wir einfach alle Variablen als global und sind sämtlicher Sorgen über den Datenaustausch enthoben.

Vor diesen Gedankengängen muß aber dringendst gewarnt werden. Globale Variablen lassen sich vielleicht noch ohne Probleme bei kürzeren Programmen (nicht länger als drei oder vier Seiten) anwenden. Bei umfangreicherem C-Quellcode sind globale Variablen aber sehr gefährlich. Da eine Änderung ihres Inhalts auch Auswirkungen auf alle anderen Funktionen hat, sind die Folgen nicht mehr überschaubar.

Diese Nebeneffekte machen das ganze Konzept des modularen Aufbaus von C zunichte und führen über kurz oder lang ins (vor-)programmierte Chaos.

C bietet viel zuverlässigere Wege zur Parameterübergabe und -übernahme. Parameter haben Sie bereits in Ihrem ersten Programm übergeben. Erinnern Sie sich an »printf(»Hallo, Welt!")«? Die Stringkonstante »Hallo, Welt!« ist nichts anderes als ein Parameter, der an die Funktion printf abgeschickt wird. C-Funktionen können eine beliebige Anzahl von Parametern beliebiger Datentypen (int, char, float und so weiter) übernehmen, zum Beispiel:

```
putchar(64);
printf("TEXT");
putc(3,6);
func(3,a,'*',b,1);
```

Die Funktionen kopieren sich die Argumente in sogenannte »formale Parameter«. Das sind nichts anderes als lokale Variablen, in die der Inhalt der angegebenen Parameter übertragen wird. Die formalen Parameter sind also nicht die Variablen selbst, sondern sie repräsentieren nur ihren Wert. Eine Veränderung ihres Wertes variiert folglich nicht die Variablen, die als Argumente in der aufrufenden Funktion benutzt werden.

Schreiben wir ein Programm, das Integer-Daten aus dem Hauptprogramm in eine Unterfunktion übernimmt und auf dem Bildschirm ausgibt:

```
main()
{
 int i; /* Parameter-
        Variable vereinbaren */
 i=4711;
 drucke(i);
 drucke(4711);
}
 drucke(x) /* x ist formaler
          Parameter*/
 int x; /* Parametertyp
        definieren */
{
 printf("%d\n",x);
}
```

Der Aufruf »drucke(i);« führt dazu, daß der Computer sich den Wert von i holt

(nicht i selbst!) und an das lokale x in drucke(x) übergibt. »drucke(4711)« weist den Computer an, die Konstante 4711 dem lokalen x zuzuweisen.

Ebenso lassen sich zwei, drei oder eine beliebige andere Zahl von Parametern übergeben:

```
main()
{
 int x,y,z;
 x=y=23;
 y++;
 z=-x;
 drucke3var(x,y,z);
}
 drucke3var(a,b,c)
 int a,b,c;
{
 printf("%d %d %d",a,b,c);
}
```

Nebenbei ist an diesem Programm die Zeile »x=y=23« recht interessant.



Dieser Befehl weist den Variablen x und y gleichzeitig den Wert 23 zu. Das läßt sich mit beliebig vielen Variablen machen: »a=b=c=d=e=f=44«. Diese Form der Wertzuweisung darf nicht mit den Logiktests in Basic verwechselt werden, wo A=B=3 die Variable B mit 3 vergleicht und abhängig davon A auf TRUE (-1) oder FALSE (0) setzt.

Ist Ihnen aufgefallen, daß die Variablendefinition in drucke3var(a,b,c) nicht, wie bisher üblich, in die geschweiften Klammern des Programmcodes integriert ist? Der Grund liegt darin, daß die zu den formalen Parametern korrespondierenden Definitionen außerhalb der Klammern stehen müssen. Die übrigen »normalen« Variablendefinitionen sind wie gehabt in die Klammern der Verbundanweisung zu setzen:

```
main()
{
 druckschleife('*',70);
 druckschleife(33,70);
 druckschleife('-',20);
}
druckschleife(zeichen,wiederhol)
char zeichen;    /* formale
                    Parameter */
int  wiederhol;
{
int zaehler;   /* normale
                  lokale
                  Variable */
for (zaehler=0; zaehler<
     wiederhol; zaehler++)
     putchar(zeichen);
putchar('\n');
}
```

Der Funktion »druckschleife(zeichen,wiederhol)« fällt die Aufgabe zu, das Zeichen »zeichen« »wiederhol«-mal zu drucken. druckschleife('*',70) gibt also 70 Sternchen auf dem Bildschirm aus. Die Variablen zeichen und wiederhol sind formale Parameter und werden deshalb außerhalb der geschweiften Klammern definiert. zaehler ist eine interne lokale Variable; die Definition erfolgt daher innerhalb der Blockklammern.

Die Datenübergabe geht nun problemlos vor sich. Anders sieht es mit der Übernahme von Werten aus der aufgerufenen in die übergeordnete Funktion aus.

Prinzipiell gibt es in C zwei Arten von Funktionsaufrufen:

- funktion(parameter);
- variable=funktion(parameter);

Im ersten Fall werden nur Daten an die Funktion übermittelt, ein Rückgabewert wird nicht erwartet. Wohin auch damit? Damit entspricht dieser Fall gewissermaßen den Prozeduren in Pascal.

Im Gegensatz dazu erwartet man beim zweiten Funktionsaufruf, daß die Unterfunktion irgendeinen Wert in die Variable einträgt. Dabei kann das return-Statement Verwendung finden.

Ein Hinweis an die Basic-Programmierer unter Ihnen: Eine der beiden Formen des return-Befehls ist identisch mit dem Basic-Kommando RETURN (Rücksprung aus Unterprogrammen).

Das normale return-Statement beendet die Ausführung einer Funktion und bewirkt die Rückkehr ins aufrufende Programm. Sie können aber auf die Anwendung verzichten, da die Rückkehr sowieso von der geschlossenen geschweiften Klammer ausgelöst wird:

```
main()
{
 funktion();
}
 funktion();
{
 printf("Hallo, hier bin ich!");
 return; /* kann entfallen! */
}
```

Als recht nützlich erweist sich aber return, um zum Beispiel, abhängig von einer if-else- oder switch-case-Konstruktion, die Bearbeitung einer Funktion zu beenden:

```
main()
{
 int i;
 i=getchar(); /* Tastaturzeichen
                 lesen */
 check_small(i); /* Pruefen, ob
         zwischen 'a' und 'z' */
}
 check_small(x)
 int x;
{
 if (x>='a' && x<='z') { printf
      ("Liegt zwischen a und z");
                           return;
}
 printf("Kein Kleinbuchstabe!");
}
```

Das Programm liest ein Zeichen von der Tastatur und prüft mit der Funktion check_small, ob es sich um einen Kleinbuchstaben handelt. In diesem Fall gibt der Computer »Liegt zwischen a und z« aus und bricht die Bearbeitung der Funktion check_small ab. Andernfalls meldet er »Kein Kleinbuchstabe!« und beendet ebenfalls die Funktion.

Die erweiterte Form return(x) schickt den Inhalt der Variablen x an die Variable, die beim Funktionsaufruf zu finden ist. Eine umständliche Formulierung für die Zuweisung »i=5« sieht in C so aus:

```
main()
{
 int i;
 i=zuweisfunkt();
 printf("%d",i);
}
 zuweisfunkt()
{
 return(5); /* 5 an die auf-
              rufende Funktion */
}
```

»int i« weist den Computer an, i als Integervariable zu verstehen. »i=zuweisfunkt()« bewirkt den Funktionsaufruf, und printf sorgt für die Bild-

schirmausgabe, anhand derer Sie die Arbeitsweise von return überprüfen können. »zuweisfunkt()« enthält als einzigen Befehl die return-Anweisung, die den konstanten Wert 5 zurückgibt.

Meistens handelt es sich wohl vor allem um die Zusammenarbeit zwischen der Übernahme und der Rückgabe von Parametern. Die normale C-Zuweisung »variable=ausdruck;« kann auf diese Weise mit einem kleinen Programm ersetzt werden, durch »variable=calc(ausdruck);«. Im Programmieralltag ist die gezeigte Routine natürlich völlig überflüssig – es sei denn, Sie wollen auf Biegen oder Brechen den Speicher füllen:

```
main()
{
 int i,j;
 i=calc(3);       /* i=3 */
 j=calc(i*2);     /* j=i*2 */
 i=calc(i+i*j/2); /* i=i+i*j/2 */
 printf("i=%d j=%d",i,j);
}
 calc(x)
 int x; /* formaler Parameter */
{
 return(x);
}
```

Es gibt noch zwei Sonderfälle, die bei der Programmierung auftreten können:

- Im Programmtext steht ein einfaches return ohne Variable oder return fehlt völlig. Der an die Variable (links vom Gleichheitszeichen) zugewiesene Wert ist dann ungültig und ergibt keinen Sinn.
- Ein Wert wird mittels return(x) zurückgegeben. Da aber die Funktion keine Zuweisungsvariable vorfindet (zum Beispiel printf("x") statt y=printf("x")), geht der Wert verloren. Er richtet dann aber nicht irgendwelche undefinierbaren Schäden an, wie man durchaus erwarten könnte, wenn ein Wert in einen nicht dafür vorgesehenen (und daher eventuell verbotenen Bereich) geschrieben wird.

Bisher sind wir stillschweigend stets davon ausgegangen, daß alle Funktionen Integer-Argumente liefern. Dem ist aber nicht unbedingt so. Sie können auch Zeichen, Fließkommawerte oder Leerargumente übergeben. Dazu werden die Funktionen ähnlich wie die Variablen deklariert.

```
funktion(x)
 char x;
{
 putchar(x);
}
```

können Sie gleichwertig schreiben:

```
int funktion(x)
 char x;
{
 putchar(x);
}
```

Immer wenn Sie keine derartige Angabe machen, nimmt der Compiler an, daß Sie eine Integer-Funktion deklarieren wollen. Eine Funktion, die einen float-(Fließkomma-)Wert an das aufrufende Programm übergibt, sieht so oder ähnlich aus:

```
float pi_wert()
{
 return(3.141592653);
}
```

Mit printf("%f",pi_wert()) läßt sich der Wert für Pi von einem anderen Programmteil aus aufrufen.

Neben den üblichen Definitionen gibt es noch »void«. Schnell im Englisch-Wörterbuch geblättert – dort stehen unter anderem als Übersetzung »leer« und »frei«. void zeigt dem Compiler an, daß die Funktion keinen Wert übermittelt, also eine »leere« Wertrückgabe hat:

```
void func(x)
 int x;
{
 putchar(x);
}
main()
{
 func(64);
}
```

Die Angabe von »void« ist aber eher für penible Programmierer gedacht und auch nicht in allen Compilern implementiert. Denselben Effekt erzielen Sie, wenn Sie kein »void« angeben und die Funktion statt dessen so programmieren, daß sie von selbst auf die Rückgabe von Werten verzichtet.

Wir haben vorhin festgestellt, daß die Position von Funktionen relativ zur Lage der main-Funktion völlig beliebig ist. Eine Einschränkung muß aber für die Fälle gemacht werden, die Funktionen ausdrücklich mit einem Typenmerkmal deklarieren.

Entdeckt der Compiler während des Übersetzens von main() den Aufruf einer von Ihnen definierten Funktion, kann er nicht erkennen, welchen Typ diese Funktion hat, und nimmt deshalb automatisch das am häufigsten gebrauchte »int« an. Ist der Funktionswert tatsächlich Integer, gibt es keine weiteren Probleme. Bei den anderen Typen wie »char«, »float« und »double« meldet der Compiler einen Fehler, weil er diese nicht mit seiner vorherigen Annahme in Einklang bringen kann.

Dieses Problem läßt sich auf zwei Wegen lösen:

- Sie stellen alle Funktionen, die nicht Integer sind, im Programmlisting irgendwo vor den Ort, an dem sie das erste Mal aufgerufen werden. Der Compiler hat dann intern einen Vermerk gespeichert, welchen Typ die Funktion hat:

```
char zeicheneingabe()
{
 return(getchar());
}
main()
{
 char tastenzeichen;
 printf("Bitte geben Sie ein
       Zeichen ein: ");
 tastenzeichen=zeicheneingabe();
 printf("Das Zeichen war ein %c",
       tastenzeichen);
}
```

Die Funktion »zeicheneingabe()« gleicht der getchar()-Funktion völlig. Sie gibt daher auch nur den Wert zurück, den sie von getchar() erhalten hat. In der Hauptfunktion main() druckt der Computer eine Aufforderung aus, ein Zeichen einzugeben.

»tastenzeichen=zeichengabe()« holt sich das Zeichen, und printf gibt es auf dem Bildschirm aus. Hätten Sie die Funktion zeicheneingabe() hinter main() gestellt, würde sich Ihr C-Compiler mit einer hämischen Fehlermeldung »bedanken«.

- Die zweite Möglichkeit ist die sogenannte »Vorwärtsdeklaration«. Diese läßt sich in eine globale und eine lokale Vorwärtsdeklaration trennen. Zuerst zur globalen: Sie setzen einfach an den Anfang Ihres Programms eine Definition des Funktionsnamens, die einer Variablendefinition ähnelt. Der Compiler erkennt daran den Typ der Funktion und kann sie richtig übersetzen:

```
char zeicheneingabe(); /* Vor-
           wärtsdeklaration */
main()
{
 char tastenzeichen;
 printf("Bitte geben Sie ein
       Zeichen ein: ");
 tastenzeichen=zeicheneingabe();
 printf("Das Zeichen war ein %c",
       tastenzeichen);
}
 char zeicheneingabe()/* Tatsäch-
     liche Funktionsdefinition */
{
 return(getchar());
}
```

In der Vorwärtsdeklaration dürfen keinerlei formale Parameter wie »char zeicheneingabe(x)« oder »char zeicheneingabe(x,'t',3)« angegeben werden. Sie endet daher immer mit leeren runden Klammern. Erst bei der eigentlichen Definition der Funktion sind die Parameter wie gehabt einzutragen:

```
char zeichenausgabe();
          /* Ohne Parameter! */
main()
{
 char x;
 x=zeichenausgabe(65);
 printf("\nASCII-Code %d",x);
}
 zeichenausgabe(z)
 /* Hier Parameter z angeben! */
 char z;
 /* Und Parameter deklarieren */
{
```

```
 putchar(z);
 return(z);
}
```

Die Vorwärtsdeklaration in der ersten Programmzeile enthält keinerlei Parameter. Erst bei der Funktionsdefinition zeichenausgabe(z) ist der Parameter angegeben. Die Funktion hat die Aufgabe, ein Zeichen auf dem Bildschirm auszugeben und dessen ASCII-Code als char-Variable an die aufrufende Funktion zurückzusenden. Dort kann er dann ausgewertet werden.

Was Sie schon bei der Besprechung der globalen Variablen gehört haben, wiederholt sich auch hier: Die globale Vorwärtsdeklaration ist nicht so empfehlenswert, wie sie auf den ersten Blick vielleicht aussieht. Besser ist es, in jeder Funktion, die die zu definierende Funktion aufruft, die Vorwärtsdeklaration einzusetzen. Das hat den Vorteil, daß Sie immer den Überblick haben, welche Funktion andere Funktionen benutzt:

```
main()
{
 char zeichenausgabe();
        /* Lokale Deklaration */
 char x;
 x=zeichenausgabe(65);
 printf("\nASCII-Code %d",x);
}
 zeichenausgabe(z)
  /* Hier Parameter z angeben! */
 char z;
  /* Und Parameter deklarieren */
{
 putchar(z);
 return(z);
}
```

Ein C-Compiler bietet Ihnen in Form eines Vorübersetzers, des »Pre-Processors«, zusätzliche Dienste an. Der #include-Befehl fügt andere C-Quellcode-Dateien in den Programmtext ein.

Der Präprozessor (so die deutsche Bezeichnung) ist eine Art Textaustausch- und Einsetzprogramm. Er hat »absolut keine Ahnung« von der Sprache C und dient nur dazu, die von Ihnen eingegebenen Quellcode-Dateien in eine vom Compiler lesbare Form zu bringen. Alle Präprozessor-Kommandos beginnen mit dem Doppelkreuz-Zeichen »#«. Sie sind nicht genormt, sondern vom Compiler abhängig. Wichtig für die alltägliche Programmierarbeit ist neben #include noch #define. Die übrigen Befehle wie #if, #ifdef und #ifndef dienen der bedingten Compilierung, die die meisten von Ihnen wahrscheinlich nie brauchen werden.

Die Anweisung #define gestattet es, Texten oder Zahlen im Programm sinnvolle Namen zu geben. Eine Zeile wie

```
umfang=2*radius*3.141592653
```

enthält die Zahl Pi als Konstante. Wenn Sie diesen Wert häufiger in Ihrem Programm brauchen, werden Sie die folgende Vereinfachung schätzen. Der Präprozessor erlaubt Ihnen, folgendes zu schreiben:

```
#define PI 3.141592653
umfang=2*radius*PI;
flaeche=radius*radius*PI;
```

Statt 3.141592653 darf also nach dem #define-Befehl synonym der Ausdruck »PI« benutzt werden. Beim Durchlaufen des Präprozessors ersetzt dieser automatisch alle »PI«s durch den Zahlenwert.

Der Präprozessor entfernt also stur die alten Bezeichnungen und fügt ebenso stur die neuen Namen ein. Immerhin ist er so intelligent, dies nicht mitten in Zeichenketten zu machen:

```
#define TEXT ERSATZTEXT
printf("TEXT");
```

Dieses Programm gibt also trotzdem »TEXT« aus und nicht etwa »ERSATZTEXT«. In alter Gewohnheit schreiben die C-Programmierer solche #define-Namen in Großbuchstaben, damit sie sich besser von Funktionsaufrufen, Variablennamen und ähnlichem abheben. Wenn sich irgendwann einmal der Wert der Konstanten ändert (bei Pi ist das kaum anzunehmen, bei anderen Daten aber durchaus möglich), müssen Sie nur im Programmkopf (dort sollten Sie alle #define's der Übersichtlichkeit halber zusammenfassen) die Werte ändern.

Die Syntax für den #define-Befehl sieht so aus:

```
#define AUSDRUCK1 AUSDRUCK2
```

Die Zeile schließt also nicht wie bei üblichen C-Anweisungen ein Strichpunkt ab. Den AUSDRUCK1 und den AUSDRUCK2 muß mindestens ein Leerzeichen oder ein TAB-Code voneinander trennen. Im allgemeinen ist es anzuraten, den zweiten Ausdruck bei #define in Klammern zu setzen, wenn dieser eine arithmetische Formel darstellt. Stellen Sie sich die Probleme bei dieser Definition vor:

```
#define ZWOELF 3*4
i=48/ZWOELF;
```

Sie erwarten hier als Ergebnis sicher eine Vier. Doch weit gefehlt! Der Compiler macht daraus »i=48/3*4«. Und das ist 64, nicht 4.

Von Klammern umgeben, wird der Ausdruck richtig bearbeitet:

```
#define ZWOELF (3*4)
```

Das wird zu:

```
i=48/(3*4)
```

Hieraus berechnet der Computer:

```
i=48/12=4
```

Hier das gewünschte Ergebnis. Sie können sich bei #define auch auf bereits bestehende Ausdrücke beziehen. Das folgende Programm definiert die Variable b gleichzeitig als »a« und »b«. Sie können alle drei scheinbaren Variablennamen benutzen. Diese bewirken auch alle dasselbe:

```
#define a b
#define c a
main()
{
 int c;
 a=33;
 b=b+1;
 printf("%d",c);
}
```

Der Compiler empfängt vom Präprozessor den Text in folgender Form:

```
main()
{
 int b;
 b=33;
 b=b+1;
 printf("%d",b);
}
```

Bei vielen Compilern geht #define aber über einen reinen Textaustausch hinaus. Sie erlauben die Angabe von Argumenten, die mitübersetzt werden:

```
#define QUADRAT(x) (x*x)
 printf("%d; %d;%d",QUADRAT(3),
 QUADRAT(3.14),QUADRAT(0));
```

In die vom Compiler erwartete Form überträgt das der Präprozessor:

```
printf("%d %d %d",3*3,3.14*3.14,
      0*0);
```

Wenn der Präprozessor jetzt noch den Programmcode optimieren kann, ersetzt er die Berechnungen durch Konstanten:

```
printf("%d %d %d",9,9.8596,0);
```

Zwischen den Makronamen und das Argument dürfen Sie nie einen Leerraum oder ein TAB-Zeichen einfügen, denn in diesem Fall erkennt der Präprozessor das Argument als zweiten Ausdruck und bringt das ganze Programm durcheinander. Statt »#define QUADRAT (x) (x*x)« müssen Sie also »#define QUADRAT(x) (x*x)« schreiben.

»Wann Makros, wann Funktionen?« wird Ihre Frage lauten. Allgemein läßt sich sagen, daß Makros zu längeren, aber schnelleren Programmen führen. Schließlich werden sie an jeder Stelle im Programmtext eingesetzt. Funktionen sind zwar langsamer bei der Programmbearbeitung, sparen aber im allgemeinen Speicherplatz.

Der Präprozessor ist ein beliebter Ansatzpunkt für den Compiler-Hersteller, zusätzliche Routinen in den Übersetzer einzufügen, ohne gleichzeitig die Kompatibilität mit anderen Compilern zu verlieren. Beispiele für zusätzliche #-Kommandos sind etwa »#list-« beziehungsweise »#list+« zum Ein- und Abschalten der Programmauflistung während des Compilierens oder »#line« zum Hinzufügen von Zeilennummern in die Listdatei.

(Martin Kotulla/hg)

Die Dimensionen von C

Kaum ein Programm, das Daten jeglicher Art verwaltet, kommt ohne Datenfelder aus. Sie sind damit unverzichtbarer Teil jeder Programmiersprache – so auch von C.

C kennt wie jede andere Programmiersprache Arrays, auf Deutsch Datenfelder. Sie werden ähnlich wie unter Basic angelegt. Diese sogenannte Dimensionierung gleicht der normalen Variablendefinition, mit dem Unterschied, daß die Dimension (Anzahl der Felder) in eckigen Klammern stehen muß. Ferner sind alle Felder explizit anzulegen.

```
int   a[23];
char  buchstabenfeld[255];
float kommazahlen[12];
```

Das erste Element trägt immer den Namen 0, das letzte einen Index (Subskript), der um eins niedriger ist als die angegebene Dimension. Ein Feld x[4] besitzt somit die Elemente x[0], x[1], x[2] und x[3]. Mit den Einzelvariablen eines Datenfeldes kann man genauso rechnen wie mit normalen Variablen. Wertzuweisungen wie »a[3]=233« sind ebenso erlaubt wie Formelausdrücke der Art »x[b]=x[b-1] *x[b+2]/3«. Felder des Typs char bieten die Simulation endlich auch Strings. Als Beispiel dafür steht das folgende Programm, das die Eingabe eines Satzes von der Tastatur erwartet und diesen dann in umgedrehter Reihenfolge wieder ausgibt. Es verwandelt beispielsweise den Satz »Lernen Sie doch C!« in die Wendung »!C hcod eiS nenreL«. Das Programm funktioniert aber bei Ihnen nur, wenn Ihr System mit einer gepufferten Ein- und Ausgabe arbeitet. Was das ist, haben wir ja schon weiter vorne dargestellt.

```
#define LINEFEED 10            /* Systembedingter LF-Code */

main()
 {
  char zeichen[255];           /* Char-Array mit 255 Elementen */
  int  i;                      /* Variable für Zählschleifen */
  char c;                      /* Speicher für gelesene Zeichen */
  c=i=0;                       /* c und i initialisieren */

  while (c!=LINEFEED)          /* Lesen, bis c=Line-Feed */
   { c=getchar();              /* Ein Zeichen lesen */
     zeichen[i]=c;             /* In Zeichenarray übertragen */
     i=i+1;                    /* Indexzähler erhöhen */
   }

  i=i-1;                       /* Zähler korrigieren */
  while (i>=0)                  /* Ausgabeschleife */
   {
     putchar (zeichen[i]);   /* Zeichen aus dem Feld ausgeben */
     i=i-1;                    /* Zähler vermindern */
   }
 }
```

Listing 1. Aus »Regen« wird »negeR«

Das Programm (Listing 1) bestimmt mit # define eine Konstante LINEFEED. Diese repräsentiert den Wert 10. Da nicht jeder Computer auch den ASCII-Code 10 als das Zeichen versteht, das die Eingabe abschließt, braucht ein Programmierer nur diese Zeile anzupassen, um die Routine auf anderen Geräten zum Laufen zu bringen.

»char zeichen[255]« erklärt das Feld zeichen zu einem Array mit 255 Buchstaben. »int i« definiert die Variable i, die als Schleifenzähler benutzt wird. Die Variable, die dem Compiler mit char c bekanntgemacht wird, nimmt jeweils ein Zeichen von der Tastatur entgegen, bevor es in dem Feld gespeichert wird.

»c=i=0« setzt die Variablen c und i in einem Arbeitsgang auf den Wert Null. C-Variablen sind nämlich nicht wie in Basic unbedingt mit Null vorbesetzt; sie enthalten den Wert, der zufällig an ihrer Speicherstelle steht.

Die erste while-Schleife liest solange Zeichen von der Tastatur ein, bis sie einen Zeilenvorschub-Code entdeckt. In der Schleife erfolgt die Zeicheneingabe mit »c=getchar()«; »zeichen[i] =c« überträgt den ASCII-Code der gedrückten Taste in das Datenfeld. »i=i+1« sorgt dafür, daß der Indexzähler auf das Datenfeld erhöht wird.

Nach der Schleife muß das Programm mit »i=i-1« den Zeigerwert korrigieren, da dieser immer am Schleifenende inkrementiert wird, was jetzt wieder rückgängig gemacht werden muß.

Die zweite while-Schleife gibt die einzelnen Zeichen des Arrays mit putchar in umgekehrter Reihenfolge aus, bis i den Wert 0 annimmt. In diesem Augenblick ist das Programm beendet.

C verzichtet bei den Arrays auf eine Überprüfung des Wertebereichs der Indexzähler. Sie können also ohne Fehlermeldung von einem Feld mit 10 Elementen das zwanzigste Element auslesen und auch einschreiben:

```
main()
  {
    int a[10];
    printf("%d",a[20]);
  }
```

Das C-Programm gibt dann einfach den Wert aus, der an der Adresse steht, die es intern aus der Indexnummer errechnete. Das wirft natürlich die Frage auf, wie C intern die Arrays abspeichert. Doch dazu erfahren Sie erst später mehr, wenn wir ausführlich auf den Umgang mit Zeigern (»pointer«) eingehen.

Denn es gibt noch einiges mehr zu sagen über die Datenfelder selbst. So sind Sie zum Beispiel nicht auf eindimensionale Arrays beschränkt. Sie können auch mit zwei- oder dreidimensionalen Feldern arbeiten. Die maximale Dimension wurde von Kernighan und Ritchie nicht exakt vorgeschrieben. Die meisten Compiler erklären sich auch mit fünf Dimensionen noch einverstanden.

Die zusätzlichen Dimensionen werden durch Anhängen weiterer Indizes in eckigen Klammern kenntlich gemacht:

```
main()
  {
    int a[2][3];
    int i,j;
    printf("i j a\n\n");
    for (i=0; i<=1; i=i+1)
      {
        for (j=0; j<=2; j=j+1)
          {
           a[i][j]=i+j;
           printf("%d %d %d\n",
           i,j,a[i][j]);
          }
      }
  }
```




Das Programm schreibt in ein (2,3)-dimensionales Integer-Datenfeld mit Hilfe zweier ineinander geschachtelter Laufschleifen jeweils die Summe der Werte von i und j. Anschließend gibt es diese deutlich sichtbar auf dem Bildschirm aus.

Genauso können Sie das mit mehr als zwei Dimensionen machen:

```
char  a[4][5][1];
float flt[10][20][30][40]);
```

In C dürfen Sie die Feldgröße, die in der Variablendefinition angegeben ist, nicht mit einer Variablen bezeichnen. Dort muß immer eine Konstante stehen, denn C erlaubt keine dynamische Verwaltung von Datenfeldern. Das Programm

```
main()
  {
    int a=5;
    int b[a];
  }
```

funktioniert also unter C nicht. Es führt immer zu einer Fehlermeldung und zum Abbruch der Compilierung.

Wir lernten vorher einen Weg kennen, Variablen bei der Definition mit einem Anfangswert zu initialisieren. Das ist auch bei den Datenfeldern möglich – mit der Einschränkung, daß auto-Felder da nicht mitmachen. Gestattet ist das logischerweise nur bei den statics. Automatic-Variablen werden ja bei jedem Funktionsaufruf neu angelegt. Da sie nach Bearbeitung der Funktion jedesmal gelöscht werden, woher soll dann der Computer beim erneuten Aufruf die alten Werte nehmen? Die Lösung ist relativ einfach.

Eine Vorbesetzung von eindimensionalen Arrays mit Werten programmieren Sie so:

```
main()
{
  static int feld[8]=
  {2,4,8,16,32,64,128,256};
  int i;
  for (i=0; i<8; i=i+1)
  printf("%d %d\n",i,feld[i]);
}
```

Das Programm setzt in die Elemente des Arrayfeldes der Reihe nach die Zahlen 2, 4, 8, 16, 32, 64, 128 und 256 ein. Die nachfolgende Ausleseschleife beweist, daß das funktioniert. Sie gibt den Schleifenzähler und das Feldelement mit dem betreffenden Index aus.

Wenn Sie keine Lust haben, nachzuzählen wieviele Werte Sie angegeben haben, um daraus den Wert in den eckigen Klammern zu erhalten, können Sie diese stumpfsinnige Arbeit auch dem Compiler überlassen. Er macht das gerne für Sie:

```
main()
  {
    static int feld[]=
    {2,4,8,16,32,64,128,256};
    int i;
    for (i=0; i<8; i=i+1)
    printf("%d %d\n",i,feld[i]);
  }
```

Die Angabe der Standardwerte ist auch bei mehrdimensionalen Feldern erlaubt. Allerdings gestaltet sich die Schreibweise etwas komplizierter:

```
static int feld[3][4]={ {1,2,3,4},
            {4,3,2,1}, {5,4,3,2}};
```

In einem komplexen Programm schaut das so aus:

```
main()
  {
    static int x[3][4]={{1,2,3,4},
                        {4,3,2,1},
                        {5,4,3,2}
                        };
    int i,j; /* Schleifenzähler */
    for (i=0; i<3; i=i+1)
      { for (j=0; j<4; j=j+1)
          printf("%d %d %d\n",
          i,j,x[i][j]);
        putchar('\n');
      }
  }
```

Es handelt sich wieder um zwei ineinander verschachtelte Schleifen, die alle Feldelemente zusammen mit den Werten von i und j ausdrucken. Nach jeweils vier Zeilen, also wenn die innere Schleife einmal abgearbeitet ist, gibt putchar (\) einen Wagenrücklauf und einen Zeilenvorschub aus, um das Ergebnis übersichtlicher zu gestalten.

Jetzt benötigen wir nur die Information, wie sich Datenfelder als Argumente an Funktionen übergeben lassen.

Da wir vorher feststellten, daß die Größe von Arrays konstant zu bleiben hat, müssen wir das bei der Übergabe an Funktionen wieder einschränken. Sollte man für jede Feldgröße etwa eine eigene Funktion schreiben? Natürlich nicht. Sie übergeben der Funktion den Namen des Feldes ohne nachfolgende eckige Klammern, und in den meisten Fällen sinnvollerweise eine Information über die Größe des Arrays. Im Funktionskopf ist der Feldname wieder ohne Klammern angegeben, bei der Definition der formalen Parameter mit leeren eckigen Klammern. Was sich jetzt furchtbar umständlich anhört, sieht in einem C-Programm sehr übersichtlich aus (siehe Listing 2).

Datenfelder groß und klein

Das Integerarray feld[20] enthält 20 Elemente, die Variable maxindex wird mit dieser Zahl geladen. Die for-Schleife versieht alle Elemente mit Werten, die von der Funktion »ausdrucken(feld)« auf den Bildschirm gebracht werden.

Auf diese Art lassen sich Datenfelder unabhängig von ihrer Größe an Funktionen übergeben. Doch eines müssen Sie unbedingt beachten. Während bei normalen Variablen deren Wert übermittelt wird (»call by value«) und anhand dieses Wertes der Computer eine neue Variable anlegt, macht der Computer bei Funktionsaufrufen von Datenfeldern keine Kopie, sondern läßt die Funktion mit dem originalen Datenfeld arbeiten. Eine Manipulation von einem Wert der Feldelemente innerhalb der Funktion wirkt sich demnach auch auf das Datenfeld im aufrufenden Programm aus. Dies wird durch das Programm in Listing 3 belegt.

Obwohl also das Datenfeld in main() und in subfunc() unterschiedliche Namen hat (»feld« beziehungsweise »array«), ist es doch ein- und dasselbe Array. In der Hauptfunktion wird das zweite Element mit der Zahl 5 initialisiert, in subfunc() verändert der Computer den Wert auf 20. Nach der Rückkehr nach main() steht jetzt auch dort der Wert 20.

```
main()
 {
   int feld[20];                /* Normale Definition des Arrays */
   int maxindex;                /* Variable für den maximalen Index */
   int i;                       /* Als Schleifenzähler */
   maxindex=20;                 /* Höchste Indexnummer angeben */
   for (i=0; i<20; i=i+1) feld[i]=255-i;
   ausdrucken(feld,maxindex); /* Angabe ohne Klammern! */
 }
ausdrucken(array,index)        /* "feld" ohne Klammern! */
 int array[];                  /* Formaler Parameter, leere Klammern*/
 int index;                    /* Formaler Parameter für Feldgröße */
 {
   int i;                      /* Lokaler Schleifenzähler */
   for (i=0; i<index; i=i+1) printf("%d\n",array[i]);
 }
```

Listing 2. Datenfelder verschiedener Größe

Wenn Sie sich schon einmal mit Maschinensprache auseinandergesetzt haben oder gar ein »Assembler-Profi« sind, wissen Sie sicher, was Zeiger (auf Englisch »pointers«) sind. Für Sie ist das folgende dann als Wiederholung gedacht. Die übrigen Leser erfahren einige interessante Neuheiten.

Stellen Sie sich eine ganz normale Integervariable vor. Mit printf können Sie ihren Wert ausgeben, mit Hilfe der verschiedenen Operatoren den Variablenwert verändern. Wenn Sie sich jetzt in die Lage des Computers versetzen, machen Sie sich klar, daß er sich ja irgendwo die Variableninhalte merken muß. Um sie auch wiederzufinden, legt er sie an einer Speicheradresse ab und verwaltet sie anhand dieser Adresse.

Ein Zeiger ist nun nichts anderes als eine Variable, die auf die Speicheradresse einer anderen Variablen hinweist. Hat Ihnen nun ein Zeiger die Adresse offenbart, können Sie den Variableninhalt indirekt beliebig manipulieren.

Sogar in verschiedenen Basic-Dialekten gibt es Funktionen, die die Adresse einer Variablen ausgeben. In MBasic und dem Basic-Interpreter des Atari ST ist das VARPTR(x), in Atari-6502-Basic ADDR(x) und im Schneider-Basic der Klammeraffe: PRINT $@a$.

Die Sprache C zeigt hier ihre Maschinennähe, denn das Zeigerkonzept wird vorzüglich unterstützt. Ganz ungefährlich ist das Arbeiten mit Zeigern aber nicht. Schnell gibt eine solche Variable eine falsche Adresse an, und das System verabschiedet sich mit allen intern gespeicherten Daten und Programmen auf Nimmerwiedersehen - beziehungsweise bis Sie die Reset-Taste drücken. Seien Sie also vorsichtig mit den Zeigern. Wenn Sie aber behutsam damit umgehen, vereinfachen sie die Programmierung oft erheblich.

Eine Sicherung bieten moderne C-Compiler gegen den allzu hemmungslosen Gebrauch von Zeigern. Die Variablen, die als Zeiger fungieren sollen, können den normalen Integervariablen nicht beliebig zugewiesen werden und bedürfen einer besonderen Art der Definition:

```
int   *integerzeiger;
char  *zeichenzeiger;
float *fliesskommazeiger;
```

Die Initialisierung ist also nur bis auf den vorangestellten Stern mit der normalen Variablendefinition identisch.

Der Stern spielt bei den Zeigeroperationen eine besondere Rolle, ebenso das kaufmännische Und-Zeichen »&«. Der Zeigertyp folgt dem Datentyp der Variablen, auf die er zeigt. Ein Zeiger auf einen Buchstaben hat also den Typ »char«, ein Zeiger auf eine Fließkommavariable den Typ »float« und so weiter.

```
main()
  {
    int array[3];
    array[2]=5;     /* In der Hauptfunktion mit 5 laden */
    printf("In main() hat array den Wert %d\n",array[2]);
    subfunc(array);
    printf("Jetzt auch in main(): array=%d",array[2]);
  }

 subfunc(feld)
  int feld[];
  {
    feld[2]=20;      /* In subfunc() mit 20 laden */
    printf("In subfunc() hat feld den Wert %d\n",feld[2]);
  }
```

Listing 3. Doppelte Datenmanipulation

Das &-Symbol liefert die Adresse einer Variablen in einer Zeigervariablen ab. Das funktioniert so:

```
main()
  {
    char zeichenvariable;
      /* normale Variable */
    char *zeichenzeiger;
      /* Pointervariable */
    zeichenvariable='#';
      /* Variable mit # laden */
    zeichenzeiger=
    &zeichenvariable;
      /* Pointer auf Variable */
    printf("%u",zeichenzeiger);
      /* Pointer ausdrucken */
  }
```

»zeichenvariable« wird als eine normale char-Variable definiert, »zeichenzeiger« als Zeiger auf eine char-Variable. Als nächstes weist das Programm der Variablen ein Symbol, das Doppelkreuz, zu. Die darauffolgende Zeile lädt einen Zeiger auf die Speicheradresse der Variablen in die Zeigervariable. Die printf-Zeile sorgt dafür, daß Sie die Adresse auch wirklich erfahren.

Die Formatoption »%u« in der printf-Funktion gibt einen arithmetischen Ausdruck in dezimaler Schreibung ohne Vorzeichen an (»unsigned«). Dies ist nötig, da manche Computer Adressen, die größer als +32767 sind, als negative Werte ansehen.

Doch was profitieren wir davon, daß wir jetzt die Adresse der Variablen kennen? Zugegeben: Noch nicht allzuviel, aber es kommt schließlich noch mehr. Der *-Operator ist die Umkehrung des &-Operators. Er liefert den Wert in der Adresse, auf die die Pointervariable zeigt.

Wenn also variable1 '#' ist und zeigergleich&variable1, dann erhalten Sie mit variable2gleich*zeiger das Zeichen »#«. Eine recht umständliche Wertzuweisung, die die Zeigermethode einbezieht, ließe sich dem C-Compiler so vorgeben:

```
main()
 {
  char zeichenvariable;
    /* normale Variable */
  char *zeichenzeiger;
    /* Pointervariable */
  char zweite_variable;
    /* zweite char-Variable */
  zeichenvariable='#';
    /* Variable mit # laden */
  zeichenzeiger=&zeichenvariable;
    /* Pointer auf Variable */
  zweite_variable=*zeichenzeiger;
    /* Adresseninhalt holen */
  putchar(zweite_variable);
    /* neue Variable ausdrucken */
 }
```

Die erste Zuweisung versieht »zeichenvariable« mit dem Zeichen »#«, die folgende Zeile legt einen Zeiger auf die Variable nach »zeichenzeiger« - wie gehabt. »zweite_variable=*zeichenzeiger« überträgt den Inhalt der Adresse, auf die der Zeiger zeigt, in die Variable. Das ist nichts anderes als das bekannte Doppelkreuz.

Brauchbar anwenden lassen sich die Zeiger zum Beispiel bei der Übernahme von Argumenten aus Funktionen. Sie erinnern sich sicherlich: Es wird nur der Wert von Variablen übergeben, anhand dem neue lokale Variablen angelegt werden. Eine Veränderung des Werts der lokalen Variablen hat keinerlei Auswirkungen auf die Variablen im aufrufenden Programm. Mit der Zeiger-Methode können Sie jetzt aber direkt die Variablen im Funktionsaufruf manipulieren. Sehen Sie das an einem Programm. Es nimmt ein Zeichen von der Tastatur an und wandelt es, wenn ein Großbuchstabe kommt, in einen kleinen um. Das soll mit Hilfe eines Funktionsaufrufes namens »lower« geschehen (siehe Listing 4).

Die Variable »zeichen« liest hier einen Buchstaben oder ein Symbol von der Tastatur ein. »zeiger« findet Verwendung als Speicher für die Variablen-

```
main()
  {
    char zeichen;                /* Zeichenvariable */
    char *zeiger;                /* Zeigervariable */
    zeichen=getchar();           /* Tastaturzeichen einlesen */
    zeiger=&zeichen;             /* Zeiger auf Buchstaben holen */
    lower(zeiger);               /* Funktionsaufruf über Zeiger */
    printf("%c\n",zeichen);      /* Geändertes Zeichen ausgeben */
  }

 lower(zeiger)                   /* Funktion "lower" */
  char *zeiger;                  /* Pointer als formaler Parameter */
  {
   char zeichen;                 /* Lokale Variable */
   zeichen=*zeiger;              /* Tatsächliches Zeichen holen */
   if (zeichen<'A') return;      /* Wenn kleiner 'A' Rücksprung */
   if (zeichen>'Z') return;      /* Wenn größer 'Z' Rücksprung */
   *zeiger=zeichen+32;           /* ASCII-Offset addieren */
  }
```

Listing 4. Aus groß mach klein

adresse. Anhand dieses Zeigers übergibt das Programm die Variable an die Funktion »lower«. Nach Beendigung der Funktion enthält »zeichen« den konvertierten Code. printf gibt den veränderten Variablenwert aus.

Die »lower«-Funktion arbeitet mit einem Zeiger als formalem Parameter. Sie definiert sich eine lokale Variable, die das Zeichen enthält. Der Computer stellt mit den beiden if-Entscheidungen fest, ob das Zeichen überhaupt ein Großbuchstabe ist. Wenn ja, addiert der Computer den Wert 32 zum ASCII-Code ('a'=97, 'A'=65, 65+32=97). Mit dem Zeigerbefehl »*zeiger=zeichen+32« bewirkt die Funktion auch die Änderung im Hauptprogramm.

Die Zeigervariablen eignen sich natürlich auch zum Rechnen. Besonders interessant sind die Inkrement- und Dekrementoperatoren. Sie zählen nicht mehr um 1 auf- oder abwärts, sondern um eine Einheit des Datentyps auf- oder abwärts. Zeiger auf char's verändern den Wert um ein Byte, Integerwerte um zwei Byte, long int's bei 32-Bit-Arithmetik vier Byte und Fließkommazahlen abhängig von der Genauigkeit zum Beispiel um sechs Byte.

Diese »Serviceleistung« von C wird am folgenden Programm deutlich:

```
main()
  {
    int  integer,*intzeiger;
    char character,*charzeiger;
    integer=16383;
    character='%';
    intzeiger=&integer;
    charzeiger=&character;
    printf("char: %u %u\n",
        charzeiger,++charzeiger);
    printf("int:  %u %u\n",
        intzeiger,++intzeiger);
  }
```

Wie das Programm zeigt, liegen die zwei Werte von »char« ein Byte, die von »int« jedoch zwei Byte auseinander.

Auch die Addition und Subtraktion von Integer-Werten ist bei Zeigern möglich. Die Werte werden hier ebenfalls auf die Datenbreite umgerechnet:

```
main()
  {
    int i,*izeiger;
    i=0;
    izeiger=&i;
    printf("1. %d\n",izeiger);
    izeiger=izeiger+3;
    printf("2. %d\n",izeiger);
  }
```

Die Differenz ist hier nicht drei Byte, sondern sechs. Dies kommt daher, daß Integers zwei Byte lang sind, und 2*3 ergibt nun einmal 6.

Zu den weiteren erlaubten Operationen mit Zeigervariablen gehören der Vergleich zweier Zeiger und die Subtraktion eines Zeigers von einem anderen. Dies ist aber nur möglich, wenn beide Zeiger demselben Datentyp angehören. Eine Subtraktion eines char- und eines float-Pointers funktioniert also nicht.

Wichtiger als bei einfachen Variablen sind die Zeiger bei Datenfeldern und Stringkonstanten. Arrays lassen sich so nicht nur durch die Angabe der Indizes bearbeiten, sondern auch durch einen direkten Zugriff:

```
main()
  {
    static char string[]="Dies
          ist ein Zeichenarray";
          char *zeiger;
    zeiger=&string[0];
    while (*zeiger != '\0')
          putchar(*zeiger++);
  }
```

Das Programm benutzt ein Zeichenarray mit dem Namen »string« und setzt einen Zeiger auf das erste Element. In einer while-Schleife gibt der Computer die Zeichen aus, bis er ein Byte mit dem Wert 0 entdeckt. Dies ist eine Besonderheit von Zeichenketten unter C. Sie werden der Reihe nach im Speicher als ASCII-Codes abgelegt. Das letzte Zeichen ist immer ein Nullbyte »\0«. Dies erleichtert die Bearbeitung von Strings über Zeiger erheblich mehr, als das Hantieren mit String-Deskriptoren in Basic.

Auf eine Kurzschreibweise sei auch noch hingewiesen. Statt »&string[0]« dürfen Sie auch einfach »string« schreiben:

```
main()
  {
    static char string[]=
          "Dies ist ein
          Zeichenarray";
          char *zeiger;
    zeiger=string;
    while (*zeiger != '\0')
          putchar(*zeiger++);
  }
```

Bevor Sie jetzt darüber räsonieren, wieso man die Schreibweise ändern darf, wollen wir Ihnen sagen, daß dahinter keine Logik steckt, sondern schlicht die Tippfaulheit der C-Entwickler.

Mit Pointer-Variablen läßt sich auch ein Datentyp String simulieren, ohne daß direkt sichtbar wird, daß Sie hier in Wirklichkeit auf Datenfelder zurückgreifen:

```
main()
  {
    char *string;
    string="Dies ist ein
          String in C!";
    printf("%s",string);
  }
```

Eine neue Option für printf wird auch gleich mit vorgestellt: »%s« druckt einen String aus, auf den ein Zeiger deutet. Die Ausgabe wird von der printf-Funktion beendet, sobald sie ein Nullbyte, den Stringbegrenzer, findet.

Wenn Sie ein komplexes Programm entwickeln, benutzen Sie darin wahrscheinlich nicht nur die grundlegenden Datentypen wie Integerzahlen und einzelne Buchstaben, sondern auch kombinierte Datentypen. Ein Programm zur Verwaltung von Personendaten erwartet beispielsweise Datensätze mit folgendem Aufbau:

1. Identifikations-Nummer der Person
2. Geschlecht
3. Familienstand
4. Einkommen beziehungsweise Gehalt
5. Zahl der Kinder

Bisher fast ausschließlich eine Domäne von Pascal (Stichwort RECORDs), macht C mit »structures«. Das Schlüsselwort »struct« leitet die Definition eines kombinierten Datentyps ein. Unsere Personendatei ließe sich so darstellen:

```
struct
{
  int  idnummer;
  /* Identifikations-Nummer */
  char geschlecht;
  /* "w" beziehungsweise "m" */
  char familienstand;
  /* "v"=verheiratet, "l"=ledig*/
  int  einkommen;
  /* Einkommen bzw. Gehalt */
  int  kinderzahl;
  /* Zahl der Sprößlinge */
} person;
```

Sie sehen, Sie können in einer Struktur gleichzeitig verschiedene Datentypen verwenden. Hier sind das beispielsweise »int« und »char«.

Die Strukturdefinition setzt sich aus dem Schlüsselwort »struct«, der öffnenden geschweiften Klammer »{«, der Liste der Strukturkomponenten (hier »char geschlecht; char familienstand; ...«), einer schließenden geschweiften Klammer »}«, dem Namen der Struktur (hier »person«) und einem abschließenden Strichpunkt zusammen.

Im folgenden können Sie einzelne Teile der Struktur ganz normal wie Variable verwenden. Die Strukturkomponenten werden durch den Namen »strukturname.komponentenname« angesprochen. Erika Mustermann hätte dann vielleicht folgende Daten für ihren neuen Personalausweis:

```
person.idnummer=1984;
person.geschlecht='w';
person.familienstand='v';
person.einkommen=2800;
person.kinderzahl=2;
```

Mit den Strukturelementen lassen sich auch arithmetische und logische Operationen durchführen – etwa nach einer Gehaltserhöhung:

```
person.einkommen=person.einkommen
      +200;
printf("Einkommen von Person %d "
      ,person.idnummer);
printf("ist %d",person.
      einkommen);
```

C bietet noch eine Besonderheit: Sie können auf einfache Art eine »Strukturmaske« definieren und diese dann auf mehrere Strukturen anwenden – vergleichbar mit Formularvordrucken – die unverändert mehrmals benutzt werden. Dem C-Compiler machen Sie das so verständlich:

```
struct personenmaske
{
  int  idnummer;
  /* Identifikations-Nummer */
  char geschlecht;
  /* "w" beziehungsweise "m" */
  char familienstand;
  /* "v"=verheiratet, "l"=ledig*/
  int  einkommen;
  /* Einkommen bzw. Gehalt */
  int  kinderzahl;
  /* Zahl der Sprößlinge */
};
```

Dies hat den Vorteil, daß Sie verschiedene Strukturen mit dem gleichen Aufbau auf einmal definieren können:

```
struct politiker
  {
    int gehalt;
    int beruf;
  };

  struct politiker helmut_kohl;
  struct politiker johannes_rau;
```

In diesem speziellen Fall ist aber folgende Befehlsfolge vorzuziehen:

```
main()
 {
   struct
    {
      int gehalt;
      int beruf;
    } helmut,johannes;

    helmut.gehalt=11200;
    helmut.beruf=1;
    johannes.gehalt=7450;
    johannes.beruf=2;
    printf("Verdient Helmut
    %d?\n",helmut.gehalt);
    printf("Und Johannes etwa
    %d?\n",johannes.gehalt);
 }
```

Die angegebenen Gehälter sind natürlich frei erfunden.

Strukturen können selbst wiederum Datenfelder und weitere Strukturen enthalten. Doch die nötigen Programmierkenntnisse gehen weit über das Ziel dieses Einführungskurses hinaus.

Wir kommen langsam zum Schluß unserer Einführung. Sie haben inzwischen die Fähigkeit, recht anspruchsvolle Programme in C zu entwickeln. Doch die bloße Kenntnis der C-Befehle ist nicht alles, was man braucht, um C zu beherrschen. Von eminenter Wichtigkeit ist auch das Wissen um die C-Bibliothek. Darin sind alle für die Arbeit mit der Sprache notwendigen Funktionen abgelegt. Obwohl die Bibliotheken der einzelnen Compilerhersteller natürlich keinerlei Normung unterliegen, zeigen sich doch überraschende Ähnlichkeiten. Der Großteil der Compiler schließt sich den Bibliotheken der Unix-Systemumgebung von Kernighan und Ritchie an. Im folgenden werden Sie eine Reihe wichtiger C-Funktionen kennenlernen, die für erfolgreiches Programmieren einfach unerläßlich sind. Am Grad der Übereinstimmung mit Ihrer Compiler-Bibliothek erkennen Sie auch die Vollständigkeit Ihres Compilers.

Zuerst zu einigen Funktionen, die arithmetische Aufgaben durchführen und für verschiedene Typenkonversionen zur Verfügung stehen:

»x=abs(y)« liest in die Variable x den absoluten Betrag von y ein. Der Betrag einer Zahl ist ihr positiver Wert. Negative Zahlen werden also mit –1 multipliziert. Positive Werte bleiben unverändert. abs ist die C-Entsprechung der ABS-Funktion von Basic und Pascal:

```
printf("%d",abs(-3));            3
printf("%d",abs(3));             3
printf("%d",abs(0));             0
```

»x=sign(y)« holt sich das Vorzeichen von y und schreibt es nach x. Zahlen kleiner als 0 liefern x=–1, die Null selbst liefert eine Null, positive Zahlen melden den Wert 1:

```
printf("%d",sign(-3));          -1
printf("%d",sign(0));            0
printf("%d",sign(3));            1
```

»x=atoi(string)« verwandelt einen String in eine Integerzahl. »atoi« heißt ausgeschrieben »convert ASCII string to integer«. Der String kann entweder als char-Array definiert sein oder als Konstante im Funktionsaufruf angegeben werden. Die Funktion arbeitet wie ASC bei den meisten Basic-Interpretern. Sie überliest alle voranstehenden Leerzeichen, TAB-Aufrufe und Newline-Codes. Sobald atoi die erste Ziffer entdeckt, beginnt die Umwandlung. Diese wird durchgeführt, bis ein Zeichen erreicht ist, das keine Ziffer mehr darstellt. Vor der Zahl dürfen auch die Vorzeichen »+« und »–« stehen. Einige Beispiele für die Benutzung von atoi:

```
printf("%d",atoi("233"));      233
printf("%d",atoi("3Text"));      3
printf("%d",atoi("Text"));       0
printf("%d",atoi("-314X"));   -314
```

Eine ganze Reihe von Funktionen dient speziell der Manipulation von Strings, die aber alle eigentlich nur Arrays des char-Typs sind. »strcat (string1,string2)« verknüpft zwei Strings, (»cat« steht dabei für »concatenate«). Das Ergebnis der Verbindung wird im Bereich des ersten Strings abgelegt. Die Funktion prüft aber nicht, ob das Datenfeld wirklich groß genug ist für die neuen Daten. Im schlimmsten Fall überschreibt der erweiterte String andere Daten oder auch den Programmcode. Als Demonstration von strcat finden Sie ein kleines C-Programm, das zwei Strings definiert und dann zusammen ausdruckt. Beachten Sie dabei, das erste Feld durch die Angabe »34« groß genug zu machen, um auch den zweiten String sicher unterzubringen:

```
main()
  {
    static char string1[34]=
         "Dies ist eine ";
    static char string2[]=
         " Stringverkettung.";
    strcat(string1,string2);
    printf("%s",string1);
  }
```

strcpy (string1,string2) hingegen kopiert den zweiten String an den Anfang des ersten (»strcpy« heißt »string copy«). Dadurch, daß das Nullbyte, das alle Strings abschließt, auch



mit übertragen wird, erscheinen bei printf nicht die eventuell überzähligen Zeichen des ersetzten Strings. Im folgenden Programm zeigt der Computer aus diesem Grund nur das Wort »C-String«:

```
main()
  {
    static char string1[]=
      "Dies ist der alte String";
    static char string2[]=
      "C-String";
    strcpy(string1,string2);
    printf("%s",string1);
  }
```

Auch hier müssen Sie der strcpy-Funktion genügend Platz zur Erweiterung des Strings zugestehen.

strcmp(string1,string2) vergleicht zwei Strings und setzt abhängig vom Ergebnis die vorangestellte Variable (»strings compare«). strcmp ähnelt der Basic-Konstruktion »VERGLEICH=(A$=B$)« oder »IF A$=B$ THEN ...«. Die Funktion liefert einen negativen Wert, wenn der erste String kleiner ist als der zweite. Entsprechend ist der positive Wert definiert. Sind beide Strings identisch, ergibt die strcmp-Funktion den Wert 0:

```
printf(" %d",strcmp("BC","A"));           1
printf(" %d",strcmp("A","BC"));          -1
printf(" %d",strcmp("gleich","gleich"));  0
```

strlen(string) ermittelt die Länge eines Strings und ist damit äquivalent zur LEN-Funktion in Basic. Leerstrings kennzeichnet der Wert 0.

```
printf(" %d",strlen("C-Sprache"));   9
printf(" %d",strlen(""));            0
```

Drei Funktionen, die die Manipulation von Einzelbuchstaben gestatten, heißen »tolower« (Umwandlung in einen Kleinbuchstaben), »toupper« (Umwandlung in einen Großbuchstaben) und »toascii« (Umwandlung in ein ASCII-Zeichen durch Löschen des siebten Bits des Codes):

```
printf(" %c",tolower("A"));   a
printf(" %c",toupper("b"));   B
printf(" %c",toascii(161));   !
```

Verschiedene Funktionen testen Buchstaben und setzen daraufhin eine Variable auf 1 (TRUE) oder 0 (FALSE):

```
ISALNUM:
printf(" %d",isalnum("F"));   1
printf(" %d",isalnum("3"));   1
printf(" %d",isalnum("#"));   0
```

isalnum heißt »is it alpha-numeric?« und wird auf TRUE gesetzt, sofern das Symbol eine Ziffer oder ein Buchstabe ist.

```
ISALPHA:
printf(" %d",isalpha("Z"));   1
printf(" %d",isalpha("3"));   0
```

isalpha (»is it an alphabetic letter?«) ist TRUE, wenn das Symbol einen Buchstaben darstellt.

```
ISDIGIT:
printf(" %d",isdigit("3"));   1
printf(" %d",isdigit("A"));   0
```

isdigit (»is it a digit?«) meldet TRUE, wenn das Symbol eine Ziffer von 0 bis 9 ist.

```
ISASCII:
printf(" %d",isascii(126));   1
printf(" %d",isascii(254));   0
```

isascii (»is it an ASCII code?«) liefert den booleschen Wert TRUE, wenn der ASCII-Code des Zeichens kleiner als 128 ist. Zwischen 0 und 127 ist der ASCII-Code nämlich standardisiert, Werte von 128 bis 255 definiert jeder Computerhersteller nach Belieben:

```
ISGRAPH:
printf(" %d",isgraph(200));   1
printf(" %d",isgraph(120));   0
```

isgraph (»is it a graphic character?«) setzt TRUE, wenn der ASCII-Code des Zeichens größer als 127 ist. Dort liegen bei den meisten Computern Grafiksymbole.

```
ISCNTRL:
printf(" %d",iscntrl(13));   1
printf(" %d",iscntrl(32));   0
```

iscntrl (»is it a control character?«) übergibt TRUE, wenn das Zeichen einen Control-Code repräsentiert und demnach im Bereich ASCII 0 bis ASCII 31 liegt.

```
ISLOWER:
printf(" %d",islower('a'));   1
printf(" %d",islower('A'));   0
```

islower (»is it a lower character?«) testet, ob das Zeichen ein Kleinbuchstabe ist. Trifft dies zu, wird das Ergebnis auf TRUE gesetzt.

```
ISUPPER:
printf(" %d",isupper('Z'));   1
printf(" %d",isupper('z'));   0
```

isupper (»is it an upper character?«) prüft, ob das Zeichen ein Großbuchstabe ist. In diesem Fall ist das Ergebnis TRUE.

```
ISPRINT:
printf(" %d",isprint('*'));   1
printf(" %d",isprint(3));     0
```

isprint (»is it a printable character?«) meldet TRUE, wenn das Zeichen auszudrucken ist. Dazu muß sein ASCII-Code zwischen 32 und 126 liegen (127 ist der Delete-Code).

```
ISPUNCT:
printf(" %d",ispunct(','));   1
printf(" %d",ispunct('7'));   0
```

ispunct (»is it a punctuation character?«) stellt fest, ob es sich um ein druckbares Zeichen handelt, das nicht gleichzeitig eine Ziffer oder ein Buchstabe ist. Das wären zum Beispiel das Komma, das Ausrufezeichen, das Dollarsymbol und der »Klammeraffe«.

```
ISSPACE:
printf(" %d",isspace('\t'));   1
printf(" %d",isspace('\n'));   1
printf(" %d",isspace(32));     1
printf(" %d",'?'));            0
```

isspace (»is it a space character?«) prüft nach, ob es sich bei dem char-Symbol um ein Leerzeichen, TAB-Code oder Newline (\) handelt. Diese Zeichen ergeben TRUE, alle anderen FALSE.

Damit wären die grundlegendsten C-Funktionen besprochen. Wenn Sie die eine oder andere nicht in Ihrer Bibliothek finden können, ärgern Sie sich nicht allzu sehr darüber. Die meisten lassen sich nämlich sehr einfach simulieren. Zum Beispiel die abs-Funktion durch

```
abs(n)
    int n;
    {
      if (n<0) return(n);
              else return(n);
    }
```

Sicher entdecken Sie aber in der Library Ihres Compilers eine Vielzahl weiterer nützlicher Funktionen, für deren Erklärung hier einfach der Platz fehlt. So haben wir beispielsweise alle Funktionen zur Dateibehandlung ausgespart. Allerdings differieren diese auch bei den verschiedenen Computern. Denken Sie allein schon an die unterschiedlichen Vorschriften zur Bildung von Dateinamen.

Wenn Ihr Interesse für C geweckt ist, und Sie auch die letzten Feinheiten der Sprache kennenlernen wollen, sollten Sie sich ein entsprechendes Fachbuch kaufen. Mit der wachsenden Begeisterung für C in Amerika und Europa schwillt auch die Bücherflut an.

Auf ein Buch sei aber hier schon hingewiesen. »Programmieren in C« enthält Informationen direkt von den C-Schöpfern Kernighan und Ritchie, sozusagen direkt von der Quelle. Bei einem Disput, was in C erlaubt ist und was nicht, kann man dieses Standardwerk bedenkenlos als Referenz heranziehen. Für Anfänger (die Sie nach diesem Kurs aber eigentlich nicht mehr sind), ist das Buch aber mit Sicherheit zu schwere Kost. Es ist eben von Systemprogrammierern für Systemprogrammierer geschrieben. Wenn Sie aber C beherrschen, werden Sie auf die glasklar dargestellten Informationen sicherlich dankbar zurückgreifen. Seltsamerweise fehlen im gesamten Buch die Bindestriche in zusammengesetzten Substantiven in den Texten – Satzfehler oder eigenartiges Sprachverständnis der Übersetzer? Jedenfalls sind manche Sätze recht schwer zu verstehen. Ansonsten ist die Übersetzung aber gut gelungen.

(Martin Kotulla/hg)

Info: Brian W. Kernighan und Dennis M. Ritchie, »Programmieren in C«, Hanser-Verlag München/Wien, 1983, ISBN 3-446-13878-1. Wer die amerikanische Originalausgabe vorzieht: »The C Programming Language«, Prentice-Hall, 1977.

Gut sortiert ist halb gewonnen

Will man verschiedene Datentypen sortieren, sind auch unterschiedliche Sortierroutinen nötig. Unser Sort-Programm in der Sprache C kann mehrere Datentypen verkraften und somit eine Datei flexibler und universeller gestalten.

Das Sortieren von Daten ist eine Aufgabe, die in einer Vielzahl von Programmen vorkommt. Es gibt mehrere Sortieralgorithmen. Da aber meistens große Datenbestände sortiert werden müssen, ist es sehr wichtig, den für die jeweilige Aufgabe optimalen Sort zu verwenden.

Leider herrscht bei vielen Programmierern der Drang, Bibliotheksfunktionen an die jeweilige Aufgabe angepaßt zu schreiben. Besser und im Endeffekt zeitsparender ist es, universell einsetzbare Funktionen zu programmieren. Hat man, wie in unserem Fall, eine universelle Sortierroutine geschrieben, so kann sie in jedes Programm problemlos eingebunden werden, ohne daß erneuter Programmieraufwand anfällt.

Zu beachten ist besonders, daß die Sortierroutine nicht nur einen bestimmten Datentyp verwenden kann, sondern flexibel gehalten ist.

Ein Beispiel hierzu: Eine Lieferantendatei soll nach der Postleitzahl sortiert werden. Das ist sehr schnell programmiert. Möchte man aber die Liste eine Woche später nach den Namen ordnen, fällt für die Sortierroutine neue Programmierzeit an. Dies kann dadurch vermieden werden, daß die Routine für beliebige Datentypen einsetzbar ist. Sie kann dann in die eigene Bibliothek aufgenommen und in neue Programme eingesetzt werden. (Heiner Etzler/hi)

```
/* Diese Sortierfunktionen enthalten nur den eigentlichen
   Sortieralgorithmus.
   Fuer eine komplette Sortierfunktion muss der Programmierer
   zwei Funktionen selbst schreiben: comp() und swap().
   comp() vollzieht den Vergleich der Datenelemente, swap()
   ist fuer das Vertauschen zustaendig.
   Beide Funktionen muessen den behandelten Datentyp unbedingt
   beruecksichtigen.
   Das Sortieren von Daten ist eine Aufgabe, die in den meisten
   Programmen vorkommt. Es lohnt sich Sortiermodule fuer den
   eigenen Bedarf zu schreiben. */



/***********************************************************************/
/*          **********  Funktion Bubble-Sort  **********             */
/***********************************************************************/



/* Die Verwendung von Zeigern erlaubt, einen datensatzunabhaengigen
   Sortieralgorithmus f}r n Datensaetze zu programmieren. */

#define void int


void bubsort(n,comp,swap)
/* An diesem Haupteingangspunkt der Funktion Bubble-Sort werden
   drei Parameter uebergeben. */

unsigned n;
/* Der erste Parameter (unsigned) ist die Anzahl zu sortierender
   Datensaetze */

int (*comp)();
/* Der zweite Parameter ist ein Zeiger auf die Funktion, die alle
   Schluessel zweier Datensaetze vergleicht und einen Ganzzahlwert
    uebergibt. */
```

```
int (*swap)();
/* Der dritte Parameter zeigt auf die Funktion, die zwei Datensaetze
   vertauscht.

{

      unsigned t;
      /* t bestimmt, ob eine Vertauschung erfolgte. */

      unsigned j;



      /* j wird in Index-Records verwendet. */

      do {
      /* In dieser Funktion wird festgestellt, ob mindestens zwei
         Records existieren. Es wird mindestens ein Vergleich gemacht. */

            t = 0;
            /* Das swap-flag(Vertauschung) erhaelt den
               Anfangswert FALSE (0). */

            for(j = 0; j < n - 1; ++j)
            /* Datensatznummern von 0 bis n-1 */

                        if((*comp)(j, j + 1) > 0) {
                        /* compare (Vergleichsfunktion) wird mit den
                           Zaehlern der beiden Datensaetze als Parameter
                           aufgerufen. */

                              (*swap)(j,j + 1);
                              /* Hier wird das Vertauschen ausgefuehrt. */

                              t = 1;
                              /* Nach einer Vertauschung wird das
                                 swap-flag auf TRUE (1) gesetzt. */
                        }
      } while(t);
        /* Der Sortiervorgang wird solange ausgefuehrt, bis keine
           Vertauschung mehr notwendig ist. */

}

/*        ********** Ende Funktion Bubble-Sort ********** */


/* Der Bubble-Sort ist einer der einfachsten Sortiermethoden. Er ist
   jedoch nicht sehr effizient, da bei jedem Schleifendurchlauf jeder
   Datensatz der zu sortierenden Records mit allen anderen verglichen
   werden muss. Bei n zu sortierenden Datensaetzen muessen n hoch 2
   Vergleichsoperationen durchgefuehrt werden - bei gro~em
   Datenbestand eine sehr zeitaufwendige Methode.
   Eine Alternative zum Bubble-Sort ist der sogenannte
                         Shell-Sort.
   Hier wird eine aus n Elementen bestehende Liste in kleinere Teil-
   listen unterteilt, die dann miteinander verglichen werden. */
```

Listing. Verschiedene Sortieralgorithmen in C realisiert

Fortsetzung Seite 100

Ergänzen Sie jetzt Ihre HAPPY COMPUTER-Sammlung

Schaffen Sie sich ein interessantes Nachschlagewerk und gleichzeitig ein wertvolles Archiv!



Kennen Sie alle »Happy Computer«-Ausgaben von 1985? Suchen Sie einen ganz bestimmten Testbericht? Oder haben Sie einen Teil eines interessanten Kurses versäumt? Suchen Sie nach einer speziellen Anwendung?

Damit Sie jetzt fehlende Hefte mit »Ihrem« Artikel nachbestellen können, finden Sie auf diesen Seiten eine Zusammenstellung aller wesentlichen Artikel der Ausgaben 01 bis 06 und 08 bis 12/85.

Und so kommen Sie schnell an die noch lieferbaren Ausgaben: Prüfen Sie, welche Ausgabe in Ihrer Sammlung noch fehlt, oder welches Thema Sie interessiert. Tragen Sie die Nummer dieser Ausgabe und das Erscheinungsjahr (z.B. 2/85) auf dem Bestellabschnitt der hier eingehefteten Bestell-Zahlkarte ein. Die ausgefüllte Zahlkarte einfach heraustrennen und Rechnungsbetrag beim nächsten Postamt einzahlen. Ihre Bestellung wird nach Zahlungseingang umgehend zur Auslieferung gebracht.

Stichwort	Titel	Seite/Ausgabe
	Aktuelles	
Computer	Amiga — ein Traumcomputer wird Wirklichkeit	9/10
	Atari: Lage gefestigt	14/11
	Der »Plus/4« ist endlich da	12/2
	Grundstein einer neuen Linie und kein zweiter PC	13/10
	Heimcomputer: Muskelschwund am Markt	11/6
	Konsequentes Chaos (Der deutsche QL)	14/10
DFÜ	Akustikkoppler für C 64	9/1
	Ascom-Koppler jetzt auch für Atari	20/8
	Ein Anschluß unter dieser Nummer (Mailbox Nummern)	169/3
	Mailboxbetrieb in den USA	22/10
	Neues DFÜ-Programm für den Spectrum	22/10
	Nullmodem zum Aufstecken	12/1
Software	Atari-Schreiber jetzt für 520 ST	14/12
	Software fast zum Nulltarif	10/1
	Träume werden wahr (Schneider-Neuheiten aus England)	9/12
	Wordstar für 199 Mark	8/9
Drucker	Mac Inker, der sparsame Drucker	12/12
Floppy	Commodore-Floppy auf Trab gebracht	9/1
	Diskettenlaufwerk für den Sharp MZ-800	12/1
	Opus, »Musik« für den Spectrum	10/6
	Quick Disk — Die Floppy-Alternative (MSX)	20/4
Erweiterung	Mini-Expansion-Box für TI 99/4A	11/1
MSX	Das Musikwunder (Yamaha CX-5)	141/2
	Der Billig-MSX von Philips kommt	80/1
	CP/M mit MSX-Computer: so geht's	141/2
	Ein komplettes System von Philips	19/8
	Flotter Dreier (Sanyo, Goldstar and Canon)	23/8
	MSX-Mix	48/3
	Tasword für MSX	16/6
	Mit dem fliegenden Teppich auf Erfolgskurs	18/10
Bücher	Bücher für den C 64	71/9
	Bücher zur DFÜ	111/3
	Bücher zu Logo	168/9
	Bücher zum Denken (KI)	120/10
	Messeberichte	
	Computer-Messe Köln: nach wie vor regional	13/9
	Die neuesten Heimcomputer (Winter-CES)	8/3
	Funkausstellung in Berlin: MSX war Trumpf	9/11
	Hacker, Krimis und Spione (Sommer-CES 1985 — Teil 2)	8/9
	Hobby-tronic und Computer-Schau	9/6
	Kampf der Kolosse (Winter-CES — Teil 1)	8/4
	Sommer-CES 1985: Weiche Welle in Chicago — Teil 1	8/8
	Software-Jackpot (Winter CES — Teil 2)	9/8
	Software-Super-Show in London (PCW-Show)	12/11
KI	Künstliche Intelligenz in Wiesbaden (AI Europe)	13/12
Musik	Musikmesse Frankfurt: Midi marschiert	22/5
	Interviews	
	David Crane (Ghostbusters Autor)	17/5
	David Snider: Der Grafik-Großmeister	14/9
	Interview mit den »Print Shop«-Machern	14/8
	Jack Tramiel (Chairman Atari)	11/2
	Kay Nishi (Vize-Präsident Microsoft)	120/6
	Hardware-Tests	
Drucker	Bewußt robust (Europrint K 6311 FT)	31/6
	Software-Tests	
Textverarb.	Ein Textprogramm, das sich lohnt (Homeword/C 64)	77/4
	Jane kontra Appleworks	143/9
	Drei Drucker im Test (STX 80, Gemini 10X, CP-80X)	16/1
	(Nachhall auf Seite 149 in 4/85) DWX 305: Schönschrift zum Niedrigpreis	16/2
	Eine heiße Verbindung (EP 22, EP 44, EXD 10)	28/5
	Kompakt und leise: Matrixdrucker OLP (Centronics)	24/1
	Regenbogenfarben — wie gedruckt (Okimate 20)	164/10
	Schön oder schnell (Horizon HX 80)	21/3
	Schöne Schrift mit schnellen Nadeln (Vergleichstest Star SR-10, Epson GX-80, Panasonic KX-1091)	137/9
	Spectrum mit starken Typen (Gabriele 9009)	126/11
	Zwei Drucker für den Schneider (NLQ 401, GP 500 CPC)	112/8
Computer	Atari 520 ST: Heißer Hit mit 32 Bit	20/6
	Chinese mit britischem Paß (Triton 64)	22/2
	Computer der dritten Art — 520 ST und C 128	22/9
	Der Musik Maestro (Yamaha CX-5)	28/4
	Der Neue: Commodore PC 128	46/8
	Der »neue« Spectrum	31/1
	Ein »Einsteiger« aus Taiwan (BIT-90)	16/2
	Enterprise ist tot — hoch lebe der Mephisto (PHC 64)	28/6
	Joyce — Schneiders Einstieg in die Welt der PCs	24/11
	Quantensprung im Schneckentempo (QL dt. Version)	180/11
	Koreaner mit Deutsch-Talent (Co-Tec/MSX)	18/3
	Schneiders neue Dimension (CPC 6128)	24/10
	Sharps Jüngster (Sharp MZ-800)	20/1
	Spectrum plus oder Spectrum minus	24/4
	SVI-X'Press — ein starkes Stück gut im Griff (MSX)	128/9
	TO7/70 und MO5E — zwei Computer, ein Konzept	133/9
	Viel Computer für wenig Geld (Schneider CPC 664)	113/6
	YC-64: Fernöstlicher Biedermann (MSX Computer)	20/2
	Wer ist wer? (Atari 520 ST+ und 260 ST)	16/12
	Wie musikalisch ist mein Heimcomputer?	148/11
	6 MSX Computer im Vergleich	124/9
Laufwerke	3-Zoll-Erfahrungen (MCD-1-Floppy für Spectrum) (Discovery/Spectrum)	22/1
	Ein ungleiches Paar (Spectrum — VIC 1541 Interface)	21/4
	Lauf, Floppy, lauf (SpeedDos plus/C64)	48/12
	Preiswertes Spectr um Floppysystem (Viscount System)	21/2
	Spectrum Diskettensystem im Plus-Look	20/3
	VC 1541 wird zur Rennfloppy	42/4
Recorder	Der Spectrum Sprinter (Datenrecorder: Sprint)	28/1
	Ein billiger Speicher für alle (Recorder MC 3810)	30/8
DFÜ	DFÜ auch mit dem TI (RS 232 für TI 99/4A)	28/8
	Kommunikation mit dem Spectrum	32/4
	Spartanisch aber gut (Ascom Akustikkoppler)	169/3
Sonstiges	Computer steuert Modelleisenbahn	176/11
	Der andere Weg (Spectrum Tastatur)	19/3
	Faszination der Technik (Fischer Technik Roboter)	44/11
	Famose Formel für den C64 (Formel 64)	40/12
	Flachbildschirm mit Schwächen (LCD für Apple IIc)	137/9
	Grafpad Supergrafik für den Spectrum	16/3
	Haltet den Dieb (Alarmanlage für C 64, VC 20)	29/1
	Ohren oder Tasten? (Voice Command Modul/C 64)	40/10
	Peripherie für MSX (Plotter, 3½-Zoll-Floppy) (Joysticks im Vergleichstest)	28/1
	Roboter, Technologie der Zukunft (Fischertechnik)	45/4
	Starker Arm für Heimcomputer (Teach Robot)	38/4
	Tafelfreuden für Grafik-Gourmets (Atari Maltafel)	14/1/
	Vom Piepmatz zum Mini-Orchester (Spectrum Sound)	16/2
	Software-Tests	
Textverarb.	Ein Textprogramm, das sich lohnt (Homeword/C 64)	77/4
	Jane Kontra Appleworks	143/9
	Jedem seine Zeitung (The Newsroom)	116/6
	Schreiben mit Schneider (Vergleichstest)	141/6
	Schreiben ohne Frust	46/2
	Star Texter: die 3-Sterne-Textverarbeitung (CPC 464)	46/9
	Textverarbeitung für jedermann (Homewriter für MSX)	137/1
Sprachen	Basic-Erweiterung zum Spartarif (Aztec Basic/C 64)	78/4
	Basicode für Spectrum	144/6
	Drei Assembler für Atari-Computer im Vergleich	30/3
	Fortschritt rückwärts (CP/M-80 Emulator für 520 ST)	138/11
	Hisoft-Pascal jetzt Microdrive-kompatibel (Spectrum)	66/2
	Logo für den Atari 520 ST	134/11
	Mallard-80-Basic — ein starkes Stück	28/11
	Maschinensprache ist keine Zauberei (CPC 464)	107/8
	Personal-Basic für den Atari 520 ST	27/9
	Prozessor-Welt von morgen: C 64 simuliert 68000	42/10
	Spezielles Spiele-Basic für den Spectrum	143/6
	Welches Basic für meinen MZ-700?	48/2
	Wolf im Schafspelz (Spectrum Simulator für C 64)	140/6
	Zwölf Farben in Mode 2 (Color Star für CPC 464)	110/6
Utilities	Das Programm, das Programme macht (Progressor)	33/8
	Disketten-Doktor für den C 128	42/12
	Quicksave für Spectrum	137/4
	SM-Kit — Das Werkzeug für Lehrling und Meister (C 64)	138/1
	Software-Knackern dazwischengepfuscht (Apple II)	27/3
Grafik	Apple-Grafik zart und fein (Dazzle Draw)	138/6
	Beeindruckend (Print Shop — Druckprogramm)	50/2
	Die Maus bringt Farbe auf den Bildschirm (Apple)	82/2
	Grafik grandios (Malprogramm Blazing Paddles)	28/3
	Koala Bilder zum Anfassen (Hardcopy-Programm)	87/2
	Mit dem Joystick programmiert (Designers Pencil)	140/6
	Schneiders Künstleratelier (Grafikmaster)	43/9
	Viel Grafik für wenig Geld (Graphics Basic und Supergrafik 64 für C 64 im Vergleich)	44/2
	Vorsicht Kamera! (Take 1, Trickfilm Designer)	126/8
DFÜ	Apple II sucht Anschluß	164/3
	Contact 64 — Die Software zum Ascom-Koppler	142/6
	Spectrum auf Draht (DFÜ Vergleichstest)	124/8
Astronomie	Spectrums Sternstunden	34/3
	Sterngucker	166/10
Lernen	Muppets an Bord (Welcome Aboart)	17/8
Musik	Man höre und staune (Sight & Sound Software/C 64)	54/6
Schach	Schach dem Commodore (Schachprogramme im Vergleich)	86/6
	Schachmatt per Telefon	166/10
	Spiele-Tests	
	Amazon	148/8
	Amazon	148/9
	Archon II, Adept	126/2
	Asylum	144/3
	Athletic Land	146/1
	A View to a Kill	169/10
	Ballblazer	167/10
	Blade of Blackpool	146/9
	Boulder Dash	126/2
	Bounty Bob strikes back	139/8
	Castle of Terror	150/8
	Cavelord	124/2
	Crazy Train	144/1
	Cyclone	182/6
	D-Bug	118/2
	Deus ex Machina	146/4
	Don't buy this	169/12
	Doomdark's Revenche	148/6
	Dorodon	142/3
	Dragonsden	124/2
	Dragonworld	149/6
	Dragonworld	146/9
	Drop Zone	150/9
	Elektro Freddy	148/1
	Elite	164/10
	Eureka	144/4
	Fahrenheit 451	146/8
	Five-a Side Football	166/10
	Formula One	140/8
	Frank Brunos Boxing	166/10
	Frankie goes to Hollywood	162/10
	Fruity Frank	145/4
	Gemstone Warrior	149/6
	Ghettoblaster	169/11
	Ghostbusters	138/3
	Ghost Chaser	170/11
	Great American Cross Country Road Race	168/11
	Gremlins	149/9
	Grog's Revenge	150/6
	Hacker	167/12
	H.E.R.O.	149/8
	Hyper Sports	149/9
	Hyper Sports I	143/3
	Jump Jet	146/9
	Karateka	146/4
	Kennedy Approach	168/12
	Knight Lore	143/3
	Macbeth	144/4
	Mail Order Monsters	146/6
	Mask of the Sun	122/2
	Mask of the Sun	146/9
	Match Day	150/6
	Mindshadow	141/8
	Mr. Do	167/10
	Monster Trivia	168/10
	Nick Faldo plays the Open	169/11
	Nightshade	169/12
	Nodes of Yesod	169/12
	On Court Tennis	150/8
	Pitfall II	148/5
	QL-Chess	181/6
	Racing Destruction Set	16/6
	Rama	148/8
	Rescue on Fractalus	168/10
	Rocket Ball	140/8
	Rockford's Riot (Bolder Dash II)	168/11
	Rock'n Bolt	139/8
	Sherlock Holmes	121/2
	Seastalker	147/1
	Serpent'll Star	142/4
	Shadowfire	181/9
	Six-Gun Shootout	149/9
	Slap Shot	181/6
	Software Star	166/11
	Spelunker	142/3
	Standing Stones	146/4
	Starion	181/9
	Summer Games II	133/8
	Super Pipeline II	141/8
	The Ancient Art of War	149/8
	The Dallas Quest	147/9
	The Dam Buster	150/9
	The Fourth Protocol	166/11
	The Hitchhiker's Guide to the Galaxy	138/4
	The Hitchhiker's Guide to the Galaxy	147/9
	The Hobbit	146/9
	The Little Computer People Projekt	170/12
	The Quill	147/6
	The Way of exploding Fist	169/10
	Tour de France	170/11
	Where in the World is Carmen San Diego	183/11
	Whistler's Brother	141/3
	White Lightning	148/1
	Winter Games	164/12
	World Championship Boxing	170/12
	Xyphus	182/6
	ZimSalaBim	141/3
	Spiele Tips	
	Abenteuer im Weltraum	182/8
	Alien 8	164/9
	Amazon	172/10
	Asylum	183/6
	Atlantis	166/8
	Aztec Challenge	147/4
	Aztec Tomb	147/4
	Aztec Tomb	183/6
	Aztec Tomb	173/10
	Beach Head	88/1
	Blade of Blackpool	164/8
	Castle of Terror	166/9
	Caverns of Khafka	183/8
	Critical Mass	166/8
	Dallas Quest	164/9
	Dark Crystal	164/6
	Death in the Caribbean	142/8
	Death in the Caribbean	166/9
	Death in the Caribbean	172/12
	Doomdark's Revenge	142/8
	Eureka	164/9
	Everyone's a Wally	173/10
	Forbidden Forest	183/9
	Forest at World's End	183/6
	Forest at World's End	184/9
	Fred	164/6
	Ghostbusters	140/3
	Ghostbusters	147/4
	Ghostbusters	182/8
	Gruds in Space	183/8
	Hampstead	172/12
	Heros of Karn	166/9
	Heros of Karn	173/12
	Hexenküche	173/12
	Hobbit	146/3
	Hulk	143/8
	Hunch Back	88/1
	Jet Set Willy	164/6
	Jewels of Babylon	164/9
	Karateka	172/12
	Knight Lore	164/6
	Lode Runner	174/11
	Lode Runner	174/12
	Mask of the Sun	173/11
	Masquerade	144/8
	Message from Andromeda	164/9
	Mindshadow	174/11
	Miner 2049er	147/4
	Pirate Adventure	126/2
	Pitfall	147/4
	Pitfall II	144/8
	Pitfall II	174/10
	Quest for Tires	164/6
	Sabre Wulf	88/1
	Sands of Egypt	173/11
	Sands of Egypt	174/12
	Schloß des Grauens	182/8
	Secret Mission	173/11
	Ship of Doom	172/12
	Sorcerer of Claymorgue Castle	183/9
	Spelunker	144/8
	Strip Poker	182/5
	Summer Games	144/8
	Summer Games	172/10
	Super Huey	171/10
	The Dallas Quest	174/10
	The Hitchhiker's Guide to the Galaxy	183/9
	The Institute	172/10
	The Institute	173/12
	The Quest	173/11
	The Witness	146/3
	Time Maschine	173/11
	Transsylvanien	166/9
	Ultima II	128/2
	Ultima II	181/8
	Ultima III	181/8
	Ulysses	166/8
	Valhalla	173/11
	Whistler's Brother	144/8
	Zeppelin	182/6
	ZimSalaBim	142/8
	Zork	172/10
	Listings	
Anwendung	Alle Neune (Jahresauswertung-Kegeln/C 64)	67/8
	(Adreßverwaltung/C 64) Besseres Basic ganz einfach Software Basic 3.0/C 64)	67/3
	Datenbank mit freiem Zugriff (C 64)	86/1
	Der Halleysche Komet kommt (MSX)	78/10
	Der Spectrum am Telefon (L.d.M./Spectrum)	80/9
	Die Mini-Textverarbeitung (Spectrum)	74/8
	Nachhall auf Seite 180 in 9/85	
	Do-it-yourself-Dateiverwaltung (Mainfile II/C 64)	63/3
	Einblick ins Innenleben (Disassembler/CPC 464)	88/8
	Eine tolle Textverarbeitung für den Schneider (464)	90/3
	Nachhall auf Seite 86 in 8/85	
	Funktionen optisch aufbereitet (VZ-200/Laser)	96/3
	Geregelte Finanzen mit dem Commodore 64	80/11
	Happysynth, der Traum jedes Musikers (L.d.M./C 64)	66/6
	Nachhall auf Seite 117 in 8/85	67/4
	Morse Decoder für Funkamateure (Spectrum)	117/11
	Nebenkostenabrechnung (C 64)	63/10
	Nachhall auf Seite 80 in 12/85 Optik mit Simons Basic (C 64)	63/3
	Programme in Reih' und Glied (C 64)	62/1
	Suchen, nein danke (Dateiverwaltung/CPC 464)	83/4
	Nachhall auf Seite 117 in 8/85	
	Transistor-Schaltungen berechnen (L.d.M./Spectrum)	81/1
	Turbo-Basic-Interpreter für Atari 800XL (L.d.M.)	81/11
Grafik	Apple IIc-HiRes-Grafik auf dem Drucker	106/1
	Bewegte Bilder auf dem C 64	48/8
	Bewegte Grafik mit drei Befehlen (CPC 464)	74/10
	Bewegung: vom Sprite zum Zeichentrick (C 64)	66/9
	Farbspielereien (Atari)	89/3
	Grafikentzerrung für Matrixdrucker (Spectrum)	108/12
	Grafik-Window bekommt Nachwuchs (C 64)	88/11
	Nachhall auf Seite 80 in 12/85	
	Grafische Impressionen (C 64)	73/8
	Grafikzauber (Apple II)	78/2
	Hires Fantasy (C 64)	88/3
	Rosetten-Grafik für den Spectrum	88/3
	Schnelle Grafik aus dem Compiler (L.d.M./C 64)	49/8
	Schnelle Sprites auf allen Apple-Computern	[illegible]
	Schöne schnelle Grafik (Grafik-Paket/C 64)	60/2
	Solar-Painter (Spectrum)	101/1
	Sprite-Editor (C 64)	82/1
	Sprites drehen ganz einfach (C 64)	68/9
	Sprites per Software (CPC 464)	74/9
	Vom Bild zum Sprite (C 64)	56/9
	Zauber der Farben mit Magic Painter (L.d.M./Atari)	63/3
	Nachhall auf Seite 88 in 8/85	
	Zaubereien auf dem Bildschirm (L.d.M./Grafik/CPC 464)	80/4
	Zeichenroutine für Kreise und Ellipsen (CPC 464)	90/5
	Zykloide für Grafiker und Mathematiker (C 64)	60/1
	Nachhall auf Seite 79 in 12/85	
	24 Farben in Grafik 0 für Atari	86/6
Spiel	Das Haus des Magiers (C 64)	83/4
	Dasher, der Volltreffer (L.d.M./C 64)	83/5
	Nachhall auf Seite 117 in 8/85	
	Der rasende Raider (C 64)	75/2
	Diamantenfieber (L.d.M./Atari 48 KByte)	58/2
	Nachhall auf Seite 86 in 8/85	
	Die Abenteuer eines rasenden Reporters (Report/C 64)	80/1
	Geröllheimer (Atari)	79/5
	Gespensterjagd im Schneider (CPC 464)	74/2
	Nachhall auf Seite 88 in 8/85	
	Hefro-Karo (Spectrum)	83/[illegible]
	Lumberjack Larrys Abenteuer in Bagdad (L.d.M/C 64)	52/1
	Kalte Zeiten (Wintry Screen/C 64)	69/[illegible]
	Kneipe zum hastigen Kellner (VC 20)	68/[illegible]
	Mit dem Apple auf die Trainerbank (Aktion Apfelsaft)	100/[illegible]

Stichwort	Titel	Seite/Ausgabe
Spiel	Mit dem Atari-Computer auf Ölsuche (Atari)	68/8
	Mit Woodshot ins Manöver ziehen	63/9
	Mücke mit Tücke (C 64)	70/1
	Musikalisches Labyrinth (C 64)	80/6
	Nachhall auf Seite 117 in 8/88	
	Moonrake (C 64)	78/6
	Nachtflug (Spectrum)	72/1
	Nachhall auf Seite 88 in 5/88 Niemandsland (C 64)	72/3
	Pokerface für 16 KByte (Spectrum)	78/6
	Psycho — die Macht des Geistes (C 64)	64/8
	Nachhall auf Seite 80 in 12/88	
	Rennfahrer mit dem Joystick (Driver/C 64)	71/4
	Rettet den letzten Baum (Insekt defense/C 64)	78/2
	SAM — der Mann von der Baustelle (L.d.M./CPC 464)	108/11
	Nachhall auf Seite 79 in 12/88	
	Schatzhöhle (Atari 800XL)	78/1
	Vorsicht Hochwasser (Aquarius/L.d.M./C 64)	84/10
	Über den Wolken (Flugplanung/C 64)	73/11
	Wortsuchspiel (Spectrum)	104/3
Tips & Tricks	AMPEL — grünes Licht für Atari-Maschinen-Programme	104/12
	Auf dem Laufenden mit einer Echtzeituhr (C 64)	70/9
	Auf Trap gebracht (CPC 464)	73/12
	Autostart für Atari	88/9
	Basic bequem (C 64)	88/2
	Basic-Compactor (Spectrum)	82/10
	Nachhall auf Seite 80 in 12/88	
	Basic-Plus: Applesoft-Basic-Erweiterung (Apple II)	98/3
	Beim C 64 piepst es	70/11
	Bilder richtig konservieren (CPC 464)	97/4
	Bildschirmtrick für den Commodore 64	70/11
	Byte-Shifter (Spectrum)	91/6
	Chain Merge endlich lauffähig (CPC 464)	91/6
	Data-Generator für Apple II	87/9
	Dateien hin- und hergerissen (Atari-IBM)	94/4
	Der neue Checksummer ist da (C 64)	84/10
	Der neue Checksummer ist da (C 64)	68/11
	Der neue Checksummer (C 64)	67/12
	Deutsche Sonderzeichen unter CP/M (CPC 464)	69/12
	Die Maltafel wird zur Maus (Atari)	98/4
	Disketten sparen (C 64)	69/9
	Nachhall auf Seite 48 in 11/88	
	Disk Help für die schnelle Hilfe (Atari)	71/8
	Disk- und DOS-Utility für alle Atari-Computer	77/10
	Drei Tricks für MSX	92/6
	Ein langes Gesicht für den C 64 (Longscreen 64)	72/11
	Fehlerhilfe mit HELP & TRACE (VC 20)	98/1
	Fensterkünstler (C 64)	71/8/
	Fortschritt für den 48 KByte-Spectrum	98/6
	Find Label (Spectrum)	88/2
	Flotte Primzahlen in Hisoft Pascal (Spectrum)	86/2
	Funktionstas. mit bel. langen Befehlsfolgen (CPC 464)	73/9
	Fußball-Manager für Commodore 64	68/8
	Grafik-Hardcopy in vierfacher Größe (C 64)	61/8
	Kostenlose Speichererweiterung (C 64)	87/10
	Kriegserklärung an Software-Diebe (Spectrum)	87/6
	Listen leicht gemacht (C 64)	84/1
	Make DATA für den Spectrum	109/3
	Maschinencode-Routinen in Basic umgesetzt (CPC 464)	78/10
	Microdrive-Aufwertung (Spectrum)	122/11
	Nachhall auf Seite 79 in 12/88	
	Mondlandung (C 64)	88/8
	Musik und Farbe (C 64)	68/12
	Neue Tricks für MSX	91/6
	Nie mehr Listingkummer mit dem Checksummer (C 64)	84/2
	Nie mehr Listingkummer mit dem Checksummer (C 64)	61/4
	Peeks und Pokes für alle Atari Computer	92/6
	Prima Werkzeug für den Programmierer (C 64)	89/8
	Programmtransfer leicht gemacht (CPC 464)	72/10
	Proportionalschrift für den Spectrum	94/6
	Protokoll auf dem Drucker (CPC 464)	78/8
	RAM-Disk für Atari 800XL	119/11
	Ran an den Userport (C 64)	72/11
	Renumber 64 (C 64)	70/8
	Nachhall auf Seite 117 in 8/88	
	Rock me Amadeus (C 64)	68/11
	RSX-Befehle ohne »@« (Schneider)	73/12
	Schilderwald (Plakatschrift/C 64)	68/10
	Schluß mit der Eintönigkeit (C 64)	70/11
	Nachhall auf Seite 80 in 12/88	
	Seidenweiche Bildschirmverschiebung (CPC 464)	73/9
	Spectrums COPY besser nutzen	83/10
	Spectrumtasten mit Funktionen belegt (Spectrum)	98/1
	Spectrum Tips & Tricks	108/12
	Sprachkurs für Commodore-Basic (C 64)	90/1
	Super-Merge für Commodore 64	84/8
	Nachhall auf Seite 160 in 8/88	
	Super-Saver (C 64)	67/10
	Statuszeile mit Uhr (Atari)	96/1
	Tasword-Umlaute (Spectrum)	87/9
	Tasword 464 mit DIN-Tastatur (CPC 464)	73/10
	Texte auch im Grafikmodus (Atari)	92/8
	Tippen mit dem Plotter (C 64)	94/1
	Tips & Tricks rund um den Schneider	77/8
	Töne aus dem Atari	96/8
	Variablendump für Atari (Atari)	83/2
	Variablen-Transfer (Spectrum)	123/11
	Verflixter Listschutz (C 64)	68/10
	Nachhall auf Seite 80 in 12/88	
	Vom Maschinencode zum Basic-Programm (C 64)	68/10
	Wie die Bilder laufen lernen (Atari)	86/2
	Zeilenakrobatik auf dem Schneider	78/8
	Zwei SCREENS im schnellen Wechsel (Spectrum)	73/8
	ZX81-Utility: Nützliches für Aufsteiger (C 64)	61/11
	30 tolle Maschinencode-Routinen (Spectrum)	98/3
	Grundlagen	
Speicher	Daten am laufenden Band	26/8
	Daten auf der schnellen Scheibe	30/8
	Floppy gegen Kassette	38/8
	Selbst geschraubt ist halb gespart	39/8
	So arbeitet das 1050-Laufwerk von Atari	36/8
	So liest und schreibt die 1541	34/8
	Speichermedium Endlosband	28/8
	Tips, Tricks und Todsünden	41/8
	Wohin in Zukunft mit Bits und Bytes	24/8
Monitore	Farbmonitore — buntes Fenster zum Computer	127/6
	Monitore: Richtig geplant, gekauft und genossen	133/6
Drucker	Blitzsaubere Schrift mit Laserlicht (Laserdrucker)	147/10
	Die »heißen« Drucker (Thermodrucker)	139/10
	Farbspiele für Farbdrucker	138/10
	Mit leisen Tönen (Tintenstrahldrucker)	128/10
	Scharfe Nadel, spitze Typen (Matrix und Typenrad)	148/10
Sprachen	Auf einen Blick: Logo-Befehle	132/2
	Befehlserweiterung für RSX (CPC 464)	34/10
	CP/M — Ein Betriebssystem	64/8
	Fenster in die Zukunft: Basic auf dem 520 ST	132/12
	Logo-Spielerei oder ernsthafte Alternative	110/1
	RSX — Maschinensprache mit Komfort	34/11
DFÜ	Begriffe aus der DFÜ	181/3
	Datenübertragung im schnellen Gleichschritt	144/11
Musik	Beethoven — Bit für Bit	182/11
	Der Weg zum Kabelorchester	187/11
Eingabe	(Joystick, Maus und Rollkugel) Grafik auf dem Tablett serviert (Grafik-Tabletts)	41/6
	Wie sag ich's meinem Computer (Tastaturen)	48/6
Sonstiges	Das Interface 1 ROM und seine Nutzung	168/4
	Der Commodore 64 kann einfach alles	89/4
	Der 16er und sein RAM	43/12
	Ein großes Abenteuer: Das Adventure	128/2
	Messen + Steuern = Regeln	148/11
	MSX — Der Standard unter der Lupe	121/9
	Schnittstellen — was sind das eigentlich	36/4
	So bauen die Spiele-Baukästen	32/8
	Ströme und Kanäle im Spectrum	136/8
	Vom Traum zum Heimcomputer (68000 Prozessor)	20/11
	Weiche Hardcopy (Schneider)	74/12
	Welcher Computer spielt am besten?	158/12
	1, 2, 3 — Kalkulieren mit der Hand ist nun vorbei	80/8
	3D-Grafik	31/6
	Allgemeine Themen	
Lernen	Der Computer — Ein moderner Trichter?	116/2
	Schule mit Computer	118/10
DFÜ	Keine Angst vor DFÜ	153/3
KI	KI: Abenteuer Denken	166/6
Sonstige	Amiga Spiele Premiere	161/12
	Bits auf Abwegen	147/11
	Computer als Briefträger	148/3
	Das Daumenkino für Heimcomputer	130/9
	Der C 64 im C 128	81/11
	Ein teures Vergnügen (DFÜ-Kosten)	164/3
	Enemy Mine: Weltraumepos mit Computergrafik	27/6
	Happy-Sportspielführer	137/9
	Heimcomputer aus zweiter Hand	142/12
	Mehr als ein Computer (Die Commodore Story)	49/4
	MSX: neuer Standard — neue Chancen	118/8
	MSX-Software: Es geht voran	126/9
	Raubkopierer gegen den Rest der Welt	128/10
	Software (fast) geschenkt	161/10
	Software-Piraterie	23/8
	Software-Volltreffer	144/12
	Software zum Spartarif	163/10
	Spiele auf der schwarzen Liste	160/11
	Vom Heimcomputer-Freak zum EDV-Spezialisten	38/2
	Vom Hobby zum Geldregen	39/2
	Vom Abenteuer, ein Abenteuer zu schreiben	42/2
	Wenn mal was schiefgeht	140/12
	Wissenswertes, Fragen und Antworten zum 128er	82/11
	Zubehör und Software — das »kleine« Geschenk	39/1/
	Zu viel Kontrolle	150/3/
	1988 — Das Jahr der Eisenbahn	164/4
	Kurse	
Logo	Teil 1: Der Einstieg für Einsteiger	40/3
	Teil 2: Die Schildkröte lernt laufen	181/4
	Teil 3: Die Schildkröte wird erwachsen	183/8
	Teil 4: Die Schildkröte wird erwachsen	134/8
Pascal	Pascal für Schüler und Lehrer	86/8
	Pascal für kluge Köpfe/Teil 1	91/9
	Pascal für kluge Köpfe/Teil 2	121/10
	Pascal für kluge Köpfe/Teil 3	124/11
Atari	Schnelle Grafik für Atari Computer	124/10
C 64	Musik mit Poke und Peek/Teil 1	64/3
	Musik mit Poke und Peek/Teil 2	53/4
	Musik mit Poke und Peek/Teil 3	86/5
	Musik mit Poke und Peek/Teil 4	52/9
	Lernen Sie Ihren Commodore 64 kennen/Teil 1	68/8
	Lernen Sie Ihren Commodore 64 kennen/Teil 2	60/6
	Lernen Sie Ihren Commodore 64 kennen/Teil 4	45/9
	Lernen Sie Ihren Commodore 64 kennen/Teil 5	68/9
	Lernen Sie Ihren Commodore 64 kennen/Teil 6	45/10
	Lernen Sie Ihren Commodore 64 kennen/Teil 7	86/11
	Ohne Fleiß kein Kreis/Teil 1	48/12
CPC 464	Kein Buch mit sieben Siegeln/Teil 1	158/8
	Kein Buch mit sieben Siegeln/Teil 2	128/6
	Kein Buch mit sieben Siegeln/Teil 4	106/8
	Kein Buch mit sieben Siegeln/Teil 5	40/9
Hardware	Zugüberwachung per Computer/Teil 1	168/4
	Zugüberwachung per Computer/Teil 2	81/8
	Zugüberwachung per Computer/Teil 3	131/6
	Basteln	
	Atari 520 ST auf Abwegen	23/12
	Bilder aus dem Weltall (Schneider)	32/12
	Dem User Port geht ein Licht auf (C 64)	64/11
	Fehler in der Spectrum Hardware	43/8
	Grün ist nicht das letzte Wort (CPC 464)	38/9
	Gute Verbindung mit dem Schneider (PIO-Interface)	28/10
	Lightshow mit dem Commodore 64	44/8
	Multitalent für den Joystickanschluß (Spectrum)	30/2
	Nachhall auf Seite 88 in 8/88	
	Nachhall auf Seite 77 in 7/88	
	Neue Geräteadresse für das 1541 Laufwerk (C 64)	62/10
	Nie wieder Angst (Alarmanlage C 64)	48/3
	PIO 8255 — Ein Experimentierinterface für Spectrum	141/9
	Nachhall auf Seite 80 in 12/88	
	Schalten und walten mit dem Atari (Schaltinterface)	114/10
	Schreiben mit Schreibmaschinenqualität (C 64)	26/8
	Schreibschutz-Schalter (Atari 810 Floppy)	24/3
	Schreibschutz-Schalter (Atari 1050 Floppy)	107/11
	Sieben auf einen Port (7 Segment Anzeige/Spectrum)	24/2
	Sparen am richtigen »Drucker-Ende« (Sinclair)	23/3
	Verbesserte Cursorsteuerung beim Spectrum	29/2
	Zwei Joysticks für ein Halleluja (CPC 464)	31/8
	Marktübersichten	
TI 99/4A	Erweiterungen zum TI 99/4A	40/1
Atari	Marktübersicht Atari	48/1
	Rund um den Atari	128/11
Atari ST	Jede Menge Software	132/11
Spectrum	Anschluß gesucht: Peripherie für ZX81 und Spectrum	48/1
C 64	Interfaces für den Commodore 64	48/1
	Der Computer mit dem großen Zubehör	56/4
CPC 464	Rund um den Schneider	48/8
Akustikkoppler	Akustikkoppler, preiswert wie noch nie	160/3
Drucker	Druckerparade	129/10
	Nachhall auf Seite 80 in 12/88	
Monitore	Marktübersicht Monitore	136/8
	Nachhall auf Seite 80 in 12/88	
Musik	Musiksoftware	181/11
Grafik	Punkt,Punkt,Komma,Strich (Grafikprogramme)	42/6
Software	Softladen (Die neusten Programme und ihre Preise)	32/1
	So viel Software (Heimsoftware für Heimcomputer)	180/12
Spiele	Spiele aus dem Baukasten (Construction Sets)	38/8
Computer	Welcher Computer zum Weihnachtsfest?	136/12
	Wettbewerbe	
Aufruf	Aktion Apfelsaft	29/1
Auflösung	Bildergalerie	106/1
Auflösung	Bildergalerie (Nachlese)	142/2
Aufruf	Bithoven-Festival	46/3
Auflösung	Bithoven-Festival	128/8
Aufruf	Der Computer als Steuermann	48/11
Aufruf	Der ideale Heimcomputer	126/6
Aufruf	Der schönste Titel von 1984	108/1
Auflösung	Der schönste Titel von 1984	138/8
Auflösung	Diskettenwettbewerb	176/10
Aufruf	Happy Computer Leserwettbewerb	130/8
Auflösung	Happy Computer Leserwettbewerb	20/12
Aufruf	Ihr Einsatz (Die beste Anwendung)	104/1
Auflösung	Ihr Einsatz	144/6
Aufruf	Kunst in der Kneipe	157/9
Aufruf	Leserumfrage — Taschenrechner	70/10
Aufruf	Probleme auf der Wörmalm	179/11
Auflösung	Spiel des Jahres	148/4
Aufruf	Steno mit dem Computer	41/6
Aufruf	Was steuern, wie regeln?	46/11
Auflösung	Wer gewinnt den goldenen Besenstiel	172/11
	Leserforum	
	Atari-Tips	102/1
	Autostart für VC 20	103/1
	Basicode-2 für MZ-700	77/2
	Basic-Speicher ohne Boden (C 64)	188/11
	Basic und HiRes-Grafik (C 64)	160/12
	Commodore-Ecke	117/10
	Eingabezeile beim Spectrum speichern	110/3
	Für Atari-Fans	94/8
	Gedächtnislücke beim ZX 81	38/4
	gtext 64 an RX 80 angepaßt	103/1
	Joystickprobleme beim VC 20	77/2
	LPRINT IO — Fehlerloses Drucken auch ohne EPROM	168/12
	Probleme mit den langen Zeilen (C 64)	188/11
	Probleme mit 800XL	189/12
	Sprite-Kollision (C 64)	180/12
	Stereo aus dem Commodore 64	110/3
	Tip für Oric 1	103/1
	Unvollständige Adresse beim ZX 81	77/2
	VC 20 und Videokamera am Monitor	103/1

```
/**************************************************************************/
/*                   **********  Shell-Sort  **********                  */
/**************************************************************************/


#define void int


void shsort(n,comp,swap)
/* Haupteinstiegspunkt Shell-Sort mit der Uebergabe von drei
   Parametern. */

unsigned n;
/* Anzahl der zu sortierenden Datensaetze */

int (*comp)();
/* Zeiger auf die Funktion, die zwei Datensatzschluessel
   vergleicht. */

int (*swap)();
/* Zeiger auf die Funktion, die zwei Records vertauscht. */
{
int m;
/* Intervallanzeiger */

int h, i, j, k;
/* Zaehler (intern) */


m = n;
/* Anfangswert f}r Intervall wird gesetzt */


while(m /= 2) {
/* Ist m /= 2 gleich Null, so ist der Sortiervorgang
   beendet. */

      k = n - m;
      /* k wird initialisiert. */

      j = 1;
      /* j wird initialisiert. */


      do {
      /* In dieser Schleife wird der untere Segment-
         grenzwert gesetzt. */

         i = j;
         /* i wird initialisiert. */


         do {
         /* Mit dieser Schleife wird der untere
            Segmentgrenzwert gesetzt. */

            h = i + m;
            /* Mittleren Grenzwert setzen. */
```

```
         if((*comp)(i - 1, h - 1) > 0 {
         /* Zeiger auf die Vergleichsfunktion setzen. */

            (*swap)(i - 1, h - 1);
            /* swap wird nach Vergleich aufgerufen. */

            i -= m;
            /* Wurde vertauscht, wird hier i gesetzt. */

     } else

            break;
            /* Ausstieg aus der Schleife */

} while(i >= 1);
  /* Uberpruefen von i auf weiteren Durchlauf */

j += 1;
/* j wird f}r weitere Vergleiche inkrementiert. */

} while(j <= k);


                  /* Ende aeussere Schleife */




         }

}

/* **********  Ende Funktion Shell-Sort  *********




/* Bubble-Sort und Shell-Sort sind fuer fast alle Sortierprobleme
   die Loesung.
   Ein weiterer bekannter Sortieralgorithmus ist der sogenannte
   Quick-Sort. Dieser Sort hat eine noch hoehere Leistungsfaehigkeit
   als der Shell-Sort.
   Das folgende Listing des Quick-Sort ist ein sehr schoenes Beispiel
   f}r rekursive Programmierung.

*/


/*********************************************************************/
/*             **********  Quick-Sort  **********                   */
/*********************************************************************/



#define void int



static int(*_comp)(), (*_swap)();
```

Listing. Verschiedene Sortieralgorithmen in C realisiert (Fortsetzung)



```
/* Zwei statische Variablen werden dazu benutzt, die Funktionszeiger
   in einer Reihenfolge zu speichern, die es erlaubt, die Zahl
   der an die rekursive Funktion uebergebenen Parameter zu
   erniedrigen. */


void qusort(n,comp,swap)
/* Haupteinstiegspunkt f}r Quick-Sort mit der Uebergabe von drei
   Parametern. */


unsigned n;
/* n enthaelt die Anzahl der zu sortierenden Datensaetze. */


int (*comp)();
/* Zeiger auf die Vergleichsfunktion */

int (*swap)();
/* Zeiger auf die Vertauschfunktion */

{

     _comp = comp;
     /* Die statische Variable comp wird auf den Parameter
        comp gesetzt. */

     _swap = swap;

     /* Die statische Variable swap wird auf den Parameter
        swap gesetzt. */

     _quick(0, n - 1);
     /* Die rekursive statische Funktion quick wird mit einem
        unteren Grenzwert von 0 und einem oberen Grenzwert von n-1
        ausgefuehrt. */

}

/* Ende der Funktion */


static void _quick(lb, ub)

unsigned lb, ub;

{

      unsigned j;

      unsigned _rearr();


      if(lb < ub) {

              if(j = _rearr(lb, ub))

                     _quick(lb, j - 1);

              _quick(j + 1, ub);
```

Listing. Verschiedene Sortieralgorithmen in C realisiert (Schluß)

```
        }
}

/* Ende der Funktion */


static unsigned _rearr(lb, ub)
unsigned lb, ub;
{

        do {
                while(ub > lb && (*_comp)(ub, lb) >= 0)
                        ub--;
                if(ub != lb) {
                   (*_swap)(ub, lb);
                   while(lb < ub && (*_comp)(lb, ub) <= 0
                         lb++;
                   if(lb != ub)
                      (*_swap)(lb, ub);
                }
        } while(lb != ub);
        return lb;
}

/* Ende der Funktion */

/*            **********  Ende Quick-Sort  *********


/* Als Indizes wurden vorzeichenlose Variable (unsigned) zur
   Definition der Unter- und Obergrenze des abzusuchenden Bereiches
   verwendet. Variable dieses Typs erweitern den Zahlenbereich,
   der innerhalb der Funktion angewendet werden darf.

   Manche Prozessoren begreifen Arithmetik nur ohne Vorzeichen.
   Werden hier nun normale Integers (int) benutzt, wird bei der Code-
   Erzeugung automatisch ein Vorzeichen gesetzt.
   In C kann der weit groessere Teil von Schleifenkontrollvariablen
   vorzeichenlos definiert werden.
```

Rückkehr einer alten Dame: Eliza

»Eliza« gehört zu den Klassikern unter den Computerspielen: Der Computer schlüpft in die Rolle eines Psychoanalytikers. Damit es nicht in Vergessenheit gerät, finden Sie hier eine Version in der hochaktuellen Sprache C.

Die Originalversion von Eliza programmierte Joseph Weizenbaum am MIT (Massachusetts Institute of Technology in Boston, USA) ursprüglich in Lisp. Schon bald folgten Übersetzungen in Basic und Veröffentlichungen in verschiedenen Computerzeitschriften. Obwohl im Laufe der Zeit einige Vereinfachungen vorgenommen wurden, ist das Resultat doch noch interessant. Der Vorteil dieser Version ist, daß die Texte und Schlüsselwörter, mit deren Hilfe Eliza antwortet, nicht Bestandteil des Programms, sondern in einer eigenen Datei erfaßt sind. So können Sie ohne Neucompilierung Änderungen vornehmen und deren Auswirkungen studieren. Denkbar wäre so sogar eine Eliza-Version, die Deutsch spricht: Wenn Sie das Programm nur durch die Angabe des Namens »Eliza« starten, arbeitet es automatisch mit den Texten in der Datei Eliza.DAT. Wenn Sie aber beim Aufruf noch einen Dateinamen »Eliza <Texte>. DAT« angeben, werden die Texte aus dieser Datei genommen. Das Programm beenden Sie durch die Eingabe von »bye«.

Das Programm sucht zunächst in der Eingabe des Benutzers nach Schlüsselwörtern, die es kennt. Die Schlüsselwörter gleicher Bedeutung sind dabei in Gruppen zusammengefaßt, und zu jeder Gruppe gibt es eine Gruppe mit zum Schlüsselwort passenden Standard-Antwortsätzen. Wenn Eliza ein Schlüsselwort aus einer Gruppe findet, nimmt sie aus der korrespondierenden Gruppe einen beliebigen Standardantwortsatz. Je häufiger ein solches Wort typischerweise auftritt, desto länger sollte die Antwortliste sein – so wird oftmaliges Auftreten immer derselben Antwort vermieden.

Sobald ein Standardantwortsatz gefunden wurde, erfolgt eine weitere Umformung. Enthält er ein Sternchen »*«, so wird an dieser Stelle ein Nebensatz eingefügt. In der Standard-Eliza.DAT treten Sternchen nur am Ende eines Satzes auf. Sie dürfen sie aber auch in der Satzmitte verwenden.

Der Nebensatz entsteht aus der Eingabe des Benutzers. Der Eingabeteil, der dem Schlüsselwort folgt, ist Ausgangspunkt. Bevor er in die Antwort eingesetzt wird, wird er aber noch konjugiert: also Wörter wie »my« durch »your« ersetzt und umgekehrt.

Der so entstandene modifizierte Antwortsatz erscheint dann endgültig auf dem Bildschirm.

Leerzeilen in der Datei werden bei der Auswertung übergangen. Ein Schrägstrich »/« gilt bei der Auswertung ganz genau wie ein echtes Zeilenende als »logisches Zeilenende«. Die Datei Eliza.DAT ist in mehrere Gruppen unterteilt, die jeweils Zeilen mit dem Inhalt »$« voneinander trennen.

Die erste Gruppe enthält das Titelbild, das der Computer beim Start ausgibt. Die zweite Gruppe – die nur aus einer einzigen Zeile bestehen darf – präsentiert die Begrüßung, mit der das Programm erstmals eine Eingabe vom Benutzer anfordert.

Die dritte Gruppe beinhaltet paarweise Schlüsselwörter für die Konjugation eines Nebensatzes, die sich gegenseitig ersetzen. »i« muß in »you« und »you« in »i« umgesetzt werden. Deshalb bilden »i« und »you« ein Paar. Ab der vierten Gruppe bis zum Dateiende stehen paarweise jeweils eine Schlüsselwortgruppe und die dazu passende Antwortgruppe.

Beachten Sie, daß die Einhaltung dieser Regeln nicht überprüft wird. Wenn die Datei nicht richtig aufgebaut ist, »verhaspelt« sich Eliza und gibt nur noch Unsinn aus. Im schlimmsten (allerdings seltenen) Fall übersieht die Sortierfunktion sogar die Dateiendmarkierung, so daß das Programm abstürzt. Alle Schlüsselwörter und Konjugationen müssen klein geschrieben werden – sonst erkennt sie das Programm niemals.

Probleme bei der Antwort

Die Ergebnisse, die Eliza liefert, hängen stark von der Reihenfolge des Dateiaufbaus ab. Schlüsselwörter am Anfang der Tabelle haben Vorrang, unabhängig von der Stellung der gefundenen Wörter im Text. In der Regel ist es sinnvoll, Schlüsselwörter, die zu konjugierten Antworten führen, zuerst zu nennen, weil das die interessanteren Antworten ergibt. Sie können aber auch mit anderen Reihenfolgen experimentieren.

Es muß sichergestellt sein, daß in jeder Eingabe ein passendes Schlüsselwort gefunden wird. Jede Eingabe (bei Eliza auch in einer leeren!) enthält nur ein Leerzeichen. Das letzte Schlüsselwort der Tabelle ist deshalb ein einzelnes Leerzeichen (nicht verwechseln mit einer Leerzeile!).

Im Schlüsselwort »think« der Standarddatei Eliza.DAT ist die Buchstabenfolge »hi« enthalten. »hi« fungiert aber gleichzeitig als eigenes Schlüsselwort mit höherer Priorität – als Abkürzung von »hello«. Ohne weitere Vorkehrungen wäre so das Schlüsselwort »think« unauffindbar. Bestandteil des Schlüsselwortes »hi« ist deshalb ein Leerzeichen, um einen Fehlgriff auszuschließen. Ähnliche Probleme treten auch bei anderen Wörtern auf. Leerzeichen in der Datei haben also immer eine wichtige Bedeutung – Sie sollten ganz besonders darauf achten. Wenn ein Schlüsselwort nicht in der ersten Spalte beginnt, ist das ein Indiz dafür, daß ein wichtiges Leerzeichen voransteht. Leerzeichen am Ende einer Zeile werden durch Anhängen eines Schrägstriches »/« am Ende der Zeile sichtbar gemacht. Da der Schrägstrich ein logisches Zeilenende bedeutet, Leerzeichen aber beim Einlesen nicht beachtet werden, schadet das bei der Interpretation der Datei durch Eliza nichts.

Bei umfangreicheren Dateikonstrukten können bestimmte Präpositionen in mehreren Paaren der Konjugationstabelle auftreten. Wenn eine mehrfache Präposition bei der Analyse eines Paares in den Nebensatz eingefügt wurde, würde sie bei der Analyse des anderen Paares fälschlicherweise wieder entfernt und durch eine dritte ersetzt. Konjugations-Schlüsselwörter, die in dieser Weise eingesetzt werden, dürfen Sie deshalb in der Datei mit einem zusätzlichen Unterstreichungszeichen »_« versehen. Das verhindert ein erneutes Finden. Vor dem Einfügen des Nebensatzes in den gesamten Antwortsatz werden alle Unterstreichungszeichen automatisch entfernt.

Scheuen Sie sich nicht, die Datei Eliza.DAT nach eigenem Gutdünken zu

verändern (natürlich sollten Sie eine Sicherheitskopie aufbewahren!). Interessant ist, außer einer allgemeinen Erweiterung, etwa die Übersetzung der Texte ins Deutsche, mit Anpassung der Konjugationen oder ein ganz anderes Gesprächsthema. Vielleicht können Sie die Antworten auch so geschickt wählen, daß der Benutzer beeinflußt wird, über ein Thema zu sprechen, zu dem Eliza besonders viele Schlüsselwörter weiß?

Die vorliegende Version wurde unter CP/M-80 mit einem BDS C-Compiler compiliert und getestet. Mit geringen Änderungen läuft Eliza aber auch unter anderen Betriebssystemen und mit anderen Compilern. Da möglichst einfache Standardfunktionen verwendet werden, ist diese Eliza-Version auch mit den kleinsten Compilern zu verarbeiten.

Eliza paßt sich an

Anpassen müssen Sie die Aufrufe der Funktionen »fopen« und »getc« (zum Einlesen der Datei). Beim BDS C-Compiler erwartet die Funktion »fopen« beim Aufruf die Adresse des Dateinamens und die Adresse eines genügend großen Pufferspeichers. Die Funktion »getc« erwartet nur die Adresse des Pufferspeichers der gewünschten Datei. Die meisten Compiler verarbeiten aber andere Strukturen. Wenn Sie noch unerfahren sind, sollten Sie sich dazu ein einfaches Programm aus Ihrer »C-Bibliothek« ansehen, und die darin enthaltenen Dateioperationen im Eliza.C-Programm nachbauen. In der Regel lauten die neuen Versionen dann so:

»...if((fd = fopen(name, "r")) == NULL) {...«

zum Dateieröffnen und

»...getc(fd)...« oder ... »fgetc(f1)...«

zum Lesen eines Zeichens aus der Datei. Natürlich müssen Sie dann deklarieren »int * fd« und können die Deklarierung von »puffer[BUFFSIZ]« weglassen.

Sie ersparen sich graue Haare, wenn Sie Experimente zur Diskettenverwaltung immer auf einer leeren Diskette durchführen (mit Sicherheitskopie des bearbeiteten Programms auf einer anderen Diskette).

»CPMEOF« ist der Wert, der das Erreichen des Dateiendes signalisiert und heißt bei manchen Compilern einfach »EOF«, die Konstante »ERROR« manchmal kurz »ERR«. Wenn andere verwendete Konstanten in Ihrem »stdio.h« nicht definiert sind, können Sie sie auf einen beliebigen, Ihnen sinnvoll erscheinenden Wert festlegen.

Die Funktion »getns« unterscheidet sich von der Standard-Bibliotheksfunktion »gets« nur dadurch, daß beim Einlesen von der Tastatur die maximal erlaubte Zeilenlänge vorgegeben werden kann. Sie funktioniert allerdings nur unter CP/M-80. Wenn Sie keinen CP/M-Rechner haben, dürfen Sie den Aufruf »getns(eingabe, 77)« ohne Schaden einfach durch »gets(eingabe)« ersetzen. Dann müssen Sie eben auf den Komfort der automatischen Begrenzung der Zeilenlänge verzichten. Die Standard-Bibliotheksfunktion »bdos()« heißt manchmal auch »_bdos()« oder »Ubdos()« und ruft – ohne Einflußnahme des C-Systems – Betriebssystemfunktionen direkt auf.

Wenn der Compiler keine »Zeiger auf Zeiger« erlaubt, können die Variablendeklarationen »char * * name« durch »int * name« ersetzt werden. Die Bedeutung und Struktur der Variable ändert sich dadurch nicht. In der Funktion selbst können doppelte Sternchen trotzdem weiterverwendet werden. Die Anwendungen dieser Variablen brauchen deshalb nicht geändert werden. Dasselbe gilt für Zeigerarrays; »char * name[]« kann ohne Schaden ebenfalls durch »int * name« ersetzt werden.

Manche Compiler hängen an jeden puts-Befehl automatisch einen Zeilenvorschub an. Sie merken das sofort an den vielen Leerzeilen auf dem Bildschirm. In diesem Fall streichen Sie im Quelltext einfach die Zeilen mit »\n«.

Die Konstanten DATEILLAENGE, MAXZEILEN und MAXKEYS bestimmen, wieviele Bytes im Speicher für die Datei Eliza.DAT reserviert werden. Wenn Sie nur wenig Speicher haben, wählen Sie etwas kleinere Zahlen.

DATEILAENGE bestimmt in etwa die maximal erlaubte Dateilänge, hier also 20000 Byte. MAXZEILEN gibt an, wieviele Zeichen die Datei maximal haben darf. Beachten Sie jedoch, daß jede Zeile zwei Nummern benötigt. Wenn Sie also in der Datei 1000 Zeilen erwarten, müssen Sie MAXZEILEN auf den Wert 2000 definieren. MAXKEYS gibt die maximale Zahl Schlüsselwortgruppen – Antwortgruppenpaare an. Jede Schlüsselwortgruppe belegt allerdings drei Nummern! Mit einer Definition von »# define MAXKEYS 400« sind also nur etwa 130 Schlüsselwortgruppen erlaubt.

Das Innenleben einer Dame

Zum Vergleich: Die hier gedruckte Grunddatei Eliza.DAT benötigt mindestens die Werte DATEILAENGE = 4500, MAXZEILEN = 520, MAXKEYS = 100. Ohne Änderung der Daten können Sie also bis zu viermal längere Wortschatzdateien verwenden. Eine Überschreibung der zulässigen Maximallängen prüft das Programm allerdings nicht. Diese führt ohne Warnung zum Systemabsturz.

Der Hauptaufwand des Programms liegt darin, leistungsfähige Routinen zur Zeichenkettenverarbeitung zur Verfü-

```
/*
    beim Aufruf mit 'ELIZA' wird automatisch 'ELIZA.DAT' geladen. Durch
    Aufruf mit 'ELIZA <dateiname>' kann stattdessen eine andere Datei
    verwendet werden. Der korrekte Dateiaufbau und erlaubte Dateilaenge
    wird aber nicht ueberprueft !
    Die Funktion 'getns' muss an das Betriebssystem angepasst werden !
    Der Aufruf von 'fopen' in funktion 'einlesen' variiert von Compiler
    zu Compiler ===> unbedingt anpassen !!!
*/

#include "stdio.h"

#define DATEILAENGE 20000  /* maximale Dateigroesse in Bytes */
#define MAXZEILEN    2000  /* 2 mal die maximale Zeilenzahl der Datei */
#define MAXKEYS       400  /* 3 mal die maximalen Schluesselwortgruppen */

/*
    'getns' unterscheidet sich von der Standardbibliotheksfunktion 'gets  nur
    durch die Angabe der maximalen Zeilenlaenge und laueft nur unter CP/M-80.
    In anderen Systemen Funktionsrumpf einfach durch 'gets(eingabe)'ersetzen !
*/

getns(eingabe, maxlen)
char eingabe[];
int maxlen;
{
    char puffer[MAXLINE + 2];

    *puffer = maxlen;                   /* maximale Laenge in erstes Pufferbyte  */
    bdos(10, puffer);                   /* Aufruf CP/M - Zeileneditor            */
    puffer[2 + puffer[1]] = NULL;       /* tatsaechliche Laenge im 2. Pufferbyte */
    strcpy(eingabe, puffer + 2);        /* Text ab zweitem Pufferbyte            */
}

/*
    tlen ist  tatsaechliche Laenge des Strings text, slen ist Laenge von such.
    'instr' sucht in 'text' ab Position 'start' nach dem String 'such'. Wenn
    gefunden Rueckgabe der Position von 'such', sonst '0'.
*/
instr(start, text, tlen, such, slen)
```

Listing 1. Smalltalk mit Eliza

gung zu stellen. Der Aufwand lohnt sich aber. Die Routinen »getns«, »instr«, »einlesen« und »holeeing« sind sehr universell und auch für andere Programme brauchbar. Am besten fügen Sie sie in Ihre Standard-Bibliotheksfunktionen ein (kopieren Sie aber vorher Ihre Standard-Bibliothek). Da die Funktionen keine globalen Variablen benötigen, ist das ohne weiteres möglich.

Die Funktion »getns(eingabe,maxlen)« entspricht der Funktion »gets(eingabe)« mit zusätzlicher Angabe der Zeilenlänge, bei deren Überschreibung die Eingabe abgebrochen wird. Intern stützt sich die Funktion auf den CP/M-Zeileneditor. Deshalb ist sie leider nur unter CP/M-80 einsetzbar.

Die Funktion »einlesen(name,datei, descript)« liest die Datei, deren Name ab der Adresse »name« gespeichert ist, von der Diskette und legt sie im Speicher ab der Anfangsadresse »datei« (am besten ein Charakterarray) ab. Ein einziges Nullbyte ersetzt mehrfache Kombinationen von CR (Wagenrücklauf) und LF (Linefeed). Dadurch wird jede Zeile in einer eigenen Zeichenkette abgelegt, und betriebssystem- und compilerabhängige Unterschiede zwischen Textdateien werden beseitigt. Bei manchen Systemen genügt nur LF oder CR, um eine neue Zeile zu signalisieren, andere Systeme (zum Beispiel CP/M und MS-DOS) benötigen beide. Als Nebeneffekt werden bei dieser Manipulation alle Leerzeichen entfernt.

Im Programm Eliza gilt auch ein Schrägstrich »/« als Zeilenvorschubbyte. Bevor Sie diese Funktion in die Standardbibliothek aufnehmen, sollten Sie eventuell diese Abfrage entfernen. Wenn Sie mit mehreren Dateien zugleich arbeiten wollen, schließen Sie die Datei nach dem Einlesen, um den benötigten Diskettenpuffer wieder freizugeben. Sonst wird mit der Zeit das System verstopft.

Als Drittes wird der Funktion ein Integerfeld übergeben. In dieses werden während des Einlesens – analog zur Stringverarbeitung in Basic – Stringdescriptoren abgelegt. Jeweils zwei aufeinanderfolgende Feldelemente enthalten Informationen zum selben String. Das erste Feldelement eines Paares enthält die Anfangsadresse einer Zeile und das zweite dessen Länge. Um die Länge einer Zeichenkette festzustellen, braucht jetzt also nicht mehr die ganze Zeichenkette nach einem Nullbyte durchsucht zu werden. Das erste nicht mehr benötigte Element des Descriptorfeldes wird mit Null markiert. Zur Verwaltungsvereinfachung kann im Hauptprogramm das Descriptorfeld auch als Array definiert werden, dessen Elemente analog zu

```
char text[], such[];
int start, tlen, slen;
{
    int i, ende;
    char *txtp, *txt2p, *suchp, *such2p, c;

    if(slen == 1) {          /* Sonderfall 1-Zeichenstring (Geschwindigkeit! */
        c = *such;
        for(txtp = text + start - 1; *txtp != NULL; txtp++)
            if(*txtp == c)
                return txtp - text + 1;
        return 0;
    } else {
        ende = tlen - slen + 1;  /* letzte Position, an der such Platz hat */
        txtp = text + start - 1; /* Anfang nicht beachten */
        suchp = such + 1;
        for(i = start; i <= ende; i++) {  /* moegliche Anfangspos. von suc */
            if(*such == *txtp++)  /* Kontrollschleife nicht immer n|tig */
                for(such2p = suchp, txt2p = txtp; *such2p == *txt2p; txt2p++)
                    if(*++such2p == NULL)  /* wenn Ende von such, dann o.k */
                        return i;
        }
        return 0;
}

/*
    Datei von Diskette einlesen; dabei mehrfache Zeilenuebergaenge (CR, LF und
    Leerzeilen) ersetzen durch ein einziges 'NULL'. Zeichen '/' gelten als
    logische Zeilenuebergaenge. Zusaetzlich Aufbau eines
    Stringdescriptorenfeldes: je 2 aufeinaderfolgende descript-Elemente
    enthalten Adresse und Laenge einer Zeile
*/

einlesen(name, datei, descript)
char name[], datei[];
int descript[];
{
    char *dateip, puffer[BUFSIZ];     /* 'puffer' ist Datenpuffer zum Diskio */
    int c, *zeilp, len;               /* '*zeilp'  Zeiger in Descriptorenfeld*/

    if(fopen(name, puffer) == ERROR) {        /* Hier Anpassung an verwendeten*/
        puts("***** Datei '");                /* Compiler notwendig !         */
        puts(name);
        puts("' nicht gefunden *****\n");
        exit();
    }
    zeilp = descript;
    *zeilp++ = dateip = datei;                /* Adresse der ersten Zeile    */
    len = 0;                                  /* Startwert fuer Zeilenlaenge */
    while((c = getc(puffer)) != CPMEOF) {     /* Lesen, bis EOF auftritt     */
        if(c == ERROR) {
            if(*(dateip - 1) != NULL) {       /* Unerwartetes Dateiende wie  */
                *dateip++ = NULL;             /* Zeilenende behandeln        */
                *zeilp++ = len;
                zeilp++;
            }
            break;
        }
        if(c == 13 || c == 10 || c == '/') {   /*  Newlinemarkier. der Datei*/
            if(*(dateip - 1) != NULL) {        /* Leerzeilen nicht beachten */
                *dateip++ = NULL;
                *zeilp++ = len;                /* Laenge in Stingdescriptor  */
                *zeilp++ = dateip;             /* Adresse naechste Zeile     */
                len = 0;
            }
         } else {
            *dateip++ = c;                     /* 'normales' Zeichen         */
            len++;
         }
    }
    *dateip = EOF;                             /* Ende der Datei             */
    *--zeilp = 0;                              /* Auch im Descriptorfeld     */
}

/*
    Datei fuer Verwendung durch Eliza vorbereiten: Anfangsadressen der
    Begruessungszeile, der Konjugationsregeln, Schluesselwortgruppen bestimmen
    Nach jeder Schluesselwortgruppe folgt ein Satz dazupassender Antworten
*/

ordnen(descript, servus, konjfirst, keylist)
int descript[], *servus, *konjfirst, keylist[];
{
    char **zeilp;
    int *keyp, i;

    zeilp = descript;              /* Header - zeilen ueberspringen            */
    while(**zeilp != '$')          /* '$' ist Zeichen fuer Gruppenende         */
        zeilp += 2;
    *servus = zeilp + 2;           /* Begruessungszeile                        */
    zeilp += 6;
    *konjfirst = zeilp;            /* Descriptor der ersten Konjugation merken */
    while(**zeilp != '$')          /* Uebrige Konjugationen ueberspringen      */
        zeilp += 2;
    zeilp += 2;                    /* Korrektur                                */
    keyp = keylist;                /* Adressen der Schluesselgruppen in Feld   */
    i = 0;                         /* 0 = Schluesselwort, 1 = Antwort          */
    while(*zeilp != 0) {           /* bis Dateiende                            */
        *keyp++ = zeilp;           /* Start Schluessel/ Antwortgruppe          */
        if(i)                      /* 1 bedeutet Antwortgruppe                 */
            *keyp++ = zeilp;       /* Antwortgruppe doppelt ablegen (ist       */
        i = !i;                    /* noetig                                   */
```




```
        while(**zeilp != '$')    /* Uebrige Saetze der Gruppe ueberspringen  */
            zeilp += 2;
        zeilp += 2;
    }
    *keyp = 0;                   /* Ende der Schluesselworttabelle           */
}

/*
    Zeile von der Tastatur holen; dabei Rueckgabe der Zeilenlaenge, ent-
    fernung aller Apostrophe "'" aus Text und Umwandlung in Kleinbuchstaben
*/

holeeing(eingabe, einlen)
char eingabe[];
int *einlen;
{
    char *ziel, *quelle, c;

    puts(": ");                 /* Meldung: Zur Eingabe bereit              */
    getns(eingabe + 1, 77);     /* Zeile holen, maximal 77 Zeichen          */
    puts("\n");
    ziel = quelle = eingabe;
    *ziel = ' ';                /* Leerzeichen voranstellen                 */
    while((c = *quelle++) != NULL)  /* Zeichen aus Puffer bis Ende erreicht */
        if(c != '\'')    /* Wenn kein "'" wieder ablegen, aber ohne Luecken  */
            (c < 'A' || c > 'Z') ? *ziel++ = c : *ziel++ = c + 32;
    *ziel++ = ' ';              /* dabei Umwandlung in Kleinbuchstaben      */
    *ziel++ = ' ';              /* 2 Leerzeichen anfuegen                   */
    *ziel = NULL;               /* Endemarkierung                           */
    *einlen = ziel - eingabe;   /* Laenge berechnen                         */
}

/*
    Standard-Antwortsatz anhand der Eingabe bestimmen: Durchsuchen der
    Eingabe nach Schluesselwoertern; wenn gefunden, waehlen eines beliebigen
    Antwortsatzes aus der zugehoerigen Gruppe
*/

findantw(antwort, antlen, restpos, eingabe, einlen, keylist)
char *antwort[], eingabe[];
int keylist[], *antlen, *restpos, einlen;
{
    char **zeilp;
    int *keyp, pos;

    for(keyp = keylist; *keyp != 0; keyp += 3)    /* Schleife Wort-gruppe     */
        for(zeilp = *keyp; **zeilp != '$'; zeilp += 2) /* next Schluessel    */
            if(pos = instr(1, eingabe, einlen, *zeilp, *(zeilp + 1))) {
                *restpos = pos + *(zeilp + 1); /* Gefunden: Rueckgabe Pos.  */
                zeilp = *(keyp + 2);           /* hinter dem Schluesselwort */
                *antwort = *zeilp++;           /* Rueckgabe Adresse Antwort */
                *antlen = *zeilp++;            /* Rueckgabe Antwortlaenge   */
                if(**zeilp != '$') {           /* Beim naechstenmal naechste*/
                    *(keyp + 2) = zeilp;       /* passende Antwort          */
                } else {
                    *(keyp + 2) = *(keyp + 1); /* nach letzter passender    */
                }                              /* wieder bei erster beginnen*/
                return;
            }
}

/*
    Nebensatz konjugieren: In Tabelle konj stehen Paarweise Descriptoren auf
    gegenseitig auszutauschende Worte. z.B. 'your' in 'my' und umgekehrt.
*/

konjugation(neben, nlen, konj)
char neben[];
int *nlen, *konj;
{
    char temp[MAXLINE], **zeilp, *ziel, *quelle, c;
    int l, ss, ls, sr, lr, s, r, len;

    len = *nlen;                /* Anfangslaenge des Nebensatzes            */
    strcpy(temp, neben);        /* wird Schrittweise durch Ergebnsi ersetzt */
    for(zeilp = konj; **zeilp != '$'; zeilp += 4) {  /* '$'ist Tabellendende*/
        l = 1;                  /* Startposition zum suchen                 */
        ss = *zeilp;            /* Descriptoren eines Wortpaares holen      */
        ls = *(zeilp + 1);
        sr = *(zeilp + 2);      /* Adresse und Laenge des 2. Wortes         */
        lr = *(zeilp + 3);
        while( (s=instr(l,neben,len,ss,ls)) + (r=instr(l,neben,len,sr,lr))) {
            if(s < r ? s : !r) {     /* Test ob 2. Wort in 1. oder umgekehrt */
                strcpy(temp + s - 1, sr);  /* erstes Wort in temp einfuegen */
                strcpy(temp + s - 1 + lr, neben + s - 1 + ls);
                l = s + lr - 1;           /* Rueckuebersetzung sperren      */
                len = len + lr - ls;      /* neue Laenge der Konjugation    */
            } else {    /* Wenn 2. Wort zuerst gefunden, in 1. umwandeln    */
                strcpy(temp + r - 1, ss);
                strcpy(temp + r - 1 + ls, neben + r - 1 + lr);
                l = r + ls - 1;
                len = len + ls -lr;
            }
            strcpy(neben, temp);      /* altuelle Version des strings holen  */
        }
    }
    ziel = neben;             /* doppelte Leerzeichen am Zeilenanfang entfernen */
    quelle = *(neben + 1) != ' ' ? neben : neben + 1;
    while((c = *quelle++) != NULL)                     /* bis zum Zeilenende  */
```

Listing 1. Smalltalk mit Eliza (Fortsetzung)

Records in Pascal »structs« aus je zwei Integerzahlen bestehen. Diese Möglichkeit bieten allerdings nicht alle Compiler.

Die Funktion »instr(start,text,tlen, such,slen)« entspricht der gleichnamigen Basic-Funktion. Ab der Position »start« (gezählt wird hier ab 1) wird in der Zeichenkette »text« nach dem ersten Auftreten der Zeichenkette »such« gesucht. Wenn »such« gefunden wurde, wird die Position zurückgegeben, an der sie in »text« stand. Andernfalls erhält das Hauptprogramm den Wert 0. Die Suche beginnt nicht unbedingt am Anfang von »text«, sondern an der Position, die »start« angibt. Zur Geschwindigkeitssteigerung werden der Funktion mit »tlen« und »slen« auch noch die Längen von Text und Suchstring übergeben. Wenn diese Werte unbekannt sind, kann im Aufruf immer noch »strlen(text)« und »strlen(such)« statt »tlen« und »slen« verwendet werden.

Da die Routine sehr oft durchlaufen wird, spart man hier nicht an der Programmlänge, sondern lieber an der Anzahl der insgesamt auszuführenden Operationen. Wenn die Suchzeichenkette die Länge 1 hat, wird eine besonders einfache Schleife durchlaufen, die nur eine Abfrage auf »Zeichen gefunden« enthält. Ist die Suchzeichenkette länger als ein Zeichen, wird in der äußeren Schleife zunächst nur geprüft, ob das erste Zeichen der Suchzeichenkette mit einem Zeichen des Textes übereinstimmt. Erst wenn eine Übereinstimmung auftaucht, wird die Schleife aufgebaut, die feststellt, ob auch noch die folgenden Zeichen passen. Sobald ein Unterschied festgestellt wird, bricht die Schleife ab. Eine weitere Beschleunigung ist möglich, wenn Ihr Compiler Registervariablen bietet. Sogar »static«-Variablen sind noch wesentlich schneller greifbar als »automatic«-Variablen.

Die Funktion »holeeing(eingabe, &einlen)« holt eine Eingabezeile von der Tastatur. In der Integer-Variablen »einlen« wird die Anzahl der eingegebenen Zeichen zurückgemeldet. Die Routine gibt vor der Eingabe selbständig ein Eingabeprompt (analog zum »A>« in CP/M) aus, und beendet die Eingabe auf dem Bildschirm durch einen Zeilenvorschub. Vor der Rückgabe werden alle eingegebenen Zeichen in Kleinbuchstaben umgewandelt und Apostrophe »'« entfernt. Vor der Zeile stehen jeweils ein und dahinter zwei Leerzeichen.

Die Funktion »konjugiere(&antwort, &anlen,konj)« ist die komplizierteste Funktion des Programms und eine spezielle Eliza-Funktion. Die vollständige Verarbeitung eines Konjugationswort-

paares entspricht einem Durchlauf der for-Schleife. Zunächst werden die Längen und die Adressen der beiden Wörter geholt. Die while-Schleife wird so oft passiert, bis feststeht, daß weder das eine noch das andere Wort sich im Nebensatz befindet. Wenn mindestens eines der beiden gefunden wird, wird das bearbeitet, das im Nebensatz zuerst auftaucht. Der temporäre String enthält zunächst eine Kopie des Nebensatzes. Der Ersatzstring wird an die Stelle des temporären Strings geschrieben, an der das zu ersetzende Wort steht und der Rest des Nebensatzes, der erhalten bleibt, angehängt. Danach wird die neue Länge des Nebensatzes berechnet und die Startposition zum Suchen so weitergesetzt, daß das eben ausgetauschte Wort nicht noch einmal untersucht wird. Zum Schluß wird der temporäre String wieder auf den Nebensatz kopiert. Nach der vollständigen Konjugation werden eventuelle doppelte Leerzeichen vor und hinter dem Nebensatz sowie alle Unterstreichungszeichen entfernt. Zurückgegeben wird die neue Länge des Nebensatzes.

Jedem seine Eliza

Die Funktion »findantw« gibt zu einer Eingabe den passenden Standard-Antwortsatz und die Position des Nebensatzes in der Eingabe zurück. Dazu durchsucht die Funktion alle Schlüsselwortgruppen nach einer Übereinstimmung in der Eingabezeile. Für jede Schlüsselwortgruppe sind im Feld »konj« drei Einträge reserviert. Der erste ist ein Zeiger auf die Descriptoren der Schlüsselwörter, der zweite ein Zeiger auf den ersten Descriptor der dazu passenden Antworten und der dritte ein Zeiger auf die aktuelle Antwort. Sobald ein Schlüsselwort gefunden wurde, liefert die Funktion die Descriptorinhalte der aktuellen Antwort. Danach richtet sich der Zeiger auf den descriptor der nächsten Antwort. Wenn diese Antwort nicht mehr zur Gruppe gehört (Inhalt »$«!), wird wieder die erste Antwort zur aktuellen. So ist sichergestellt, daß alle zum Schlüsselwort passenden Antworten irgendwann an der Reihe sind.

Wenn Sie nun weiter interessiert sind, können Sie darangehen, das Programm auszubauen. Vielleicht statten Sie es mit einem Gedächtnis aus, so daß auch, wenn gerade kein Schlüsselwort in der Eingabe vorkommt, Eliza feststellen kann, welches Hauptthema in der Luft liegt. Oder Sie testen, wie sich eine andere Suchreihenfolge (etwa nach der Lage im Text statt nach der Reihenfolge in der Datei) auf die Antworten auswirkt.

(Helmut Tischer/hg)

```
        if(c != '_')        /* alle Unterstriche entfernen (falls bei bestimm-*/
            *ziel++ = c;   /* ten Worten Markierung fuer bereits ersetzt     */
    *ziel = NULL;           /* notwendig war                                  */
    *nlen = ziel - neben;  /* neue Zeilenlaenge zurueckgeben                 */
}

/*
    Verlassen der Eingabeschleife durch 'bye'. Ein Stern '*' in einem Antwort-
    statz veranlasst das Einfuegen eines konjugierten Nebensatzes. Der Neben-
    satz ist der Teil der Eingabe, hinter dem Schluesselwort
*/

main(argc, argv)
int argc;
char *argv[];
{
    int descript[MAXZEILEN], *keyp, keylist[MAXKEYS],
    einlen, anlen, restpos, t, npos, nlen;
    char datei[DATEILAENGE], eagabe[MAXLINE], oldeing[MAXLINE],neben[MAXLINE],
        *name, *antwort, **servus, **konj, **zeilp;

    if(argc <= 1) {                         /* Datei einlesen und ordnen        */
        name = "ELIZA.DAT";                 /* wenn keine Dateinamensangabe:    */
    } else {                                /* Standard ist 'ELIZA.DAT'         */
        name = argv[1];
    }
    einlesen(name, datei, descript);
    ordnen(descript, &servus, &konj, keylist);
    for(zeilp = descript; **zeilp != '$'; zeilp += 2) {
        puts(*zeilp);                       /* Headerzeilen anzeigen            */
        puts("\n");
    }
    puts(*servus);                          /* Begruesung von Eliza             */
    puts("\n");
    *oldeing = NULL;                        /* alte Eingabe zunaechst loeschen  */
    for(;;) {                               /* Hauptschleife des Programms      */
        do {  /* solange abfragen, bis Eingabe unterschiedlich zur letzten */
            holeeing(eagabe, &einlen);
            if(!strcmp(eagabe, " bye  "))   /* Programmabbruch                  */
                exit();
            if(t = !strcmp(eagabe,oldeing))  /* Eingabe war schon mal da        */
                puts("please don't repeat yourself !\n");
        } while(t);
        strcpy(oldeing, eagabe);            /* neue Eingabe wird alte           */
        findantw(&antwort, &anlen, &restpos, eagabe, einlen, keylist);
        if(npos = instr(1, antwort, anlen, "*", 1)) { /*Nebensatz einfuegen?*/
            strcpy(neben, " ");  /* Nebensatz beginnt hinter Schluesselwort */
            strcpy(neben + 1, eagabe + restpos - 1);
            nlen = einlen - restpos + 2;            /* Laenge des Nebensatzes */
            konjugation(neben, &nlen, konj);        /* Konjugieren            */
            strcpy(eagabe, antwort);    /* Standard-Antwort nicht zerstoeren!*/
            strcpy(eagabe + npos - 1, neben);/* Nebensatz in Kopie einfuegen*/
            strcpy(eagabe + npos - 1 + nlen, antwort + npos);
            antwort = eagabe;           /* Kopie wird zu neuer Antwort       */
        }
        puts(antwort);              /* Gesamtantwort auf Bildschirm ausgeben  */
        puts("\n");
    }
}
/* Geschafft ! */
```

Listing 1. Smalltalk mit Eliza (Schluß)

```
 /
 /
 /
          E L I Z A
          =========
 /
              - der Computer als Psychoanalytiker
 /
Wortschatz: entnommen aus Creative Computing, Juli-August 1977
Programm:   neu entwickelt im M(rz 1986 von Isar-Amper-Soft
 /
 /
     Erz(hl' doch Eliza einfach irgendetwas,
                         wozu Du gerade Lust hast.
 /
 /
$
hi, i am eliza. tell me your problem.
$
 are / am /
were /was /
 you / i /
your /my /
 ive / youve /
 im / youre /
$
can you
$
don't you believe that i can*
perhaps you would like to be able to*
you want me to be able to*
$
can i
$
perhaps you don't want to*
do you want to be able to*
$
you are
youre
$
```

Listing 2. Elizas Antworten als Datei

```
what makes you think i am*
does it please you to believe i am*
perhaps you would like to be*
do you sometimes wish you where*
$
i dont
$
don't you really*
why don't you*
do you wish to be able to*
does that trouble you?
$
i feel
$
tell me more about such feelings.
do you often feel*
do you enjoy feeling*
$
why dont you
$
do you really believe i don't*
perhaps in good time i will*
do you want me to*
$
why cant i
$
do you think you should be able to*
why can't you*
$
are you
$
why are you interestet in whether or not i am*
whould you prefer if i where not*
perhaps in your fantasies i am*
$
i cant
$
how do you know you can't*
have you tried?
perhaps you can now*
$
i am
im /
$
did you come to me because you are*
how long have you been*
do you believe it is normal to be*
do you enjoy being*
$
you /
$
we were discussing you-- not me.
oh, i*
you're not really talking about me, are you?
$
i want
$
what would it mean to you if you got*
why do you want*
suppose you soon got*
what if you never got*
i sometimes also want*
$
what
how
who
where
when
why
$
why do you ask?
does that question interest you?
what answer would please you the most?
what do you think?
are such questions in your mind often?
what is that you really want to know?
have you asked anyone else?
have you asked such questions before?
what else comes to mind when you ask that?
$
name
$
names don't interest me.
i don't care about names-- please go on.
$
cause
$
is that the real reason?
don't any other reasons come to mind?
doeas that reason explain anything else?
what other reasons might there be?
$
sorry
$
please don't apologize!
apologies are not necessary.
what feelings do you have, when you apologize?
don't be so defensive!
$
dream
$
what does that dream suggest to you?
do you dream often?
what persons appear in your dreams?
are aou disturbed by your dreams?
$
hello
hi /
$
how do you do ... please state your problem.
$
maybe
$
you don't seem quite certain.
why the uncertain tone?
can't you be more positive?
you aren't sure?
don't you know?
$
 no
$
why no*
don't say no. its always so negative.
why not?
are you sure?
why no?
$
your
$
why are you concerned about my*
what about your own*
$
always
$
can't you think of a specific example?
when?
what are you thinking of?
really, always?
$
think
$
do you really think so?
but you are not sure you*
do you doubt you*
in what way?
$
alike
$
what resemblance do you see?
what does the similarity suggest to you?
what other connections do you see?
could there really be some connections?
how?
you seem quite positive.
are you sure?
$
yes
$
i see.
i understand.
$
friend
$
why do you bring up the topic of friends?
do your friends worry you?
do your friends pick on you?
are you sure you have any friends?
do you impose on your friends?
perhaps your love for friends worries you.
$
computer
$
do computers worry you?
are you talkink about me in partikular?
are you frightened by machines?
why do you mention computers?
what do you think machines have to do with your problems?
don't you think computers can help people?
what is it about machines that worries you?
$
money
$
why do you have problems with money?
do you think money is everything?
are you sure that money is the problem?
$
eliza
$
i think we want to talk about you, not about me.
whats about me?
why do you always bring up my name?
$
 /
$
say, do you have any psychological problems?
what does that suggest to you?
i see.
i'm not sure i understand you fully.
come elucidate your thoughts.
can you elaborate on that?
that is quite interesting.
$
```

Listing 2. Elizas Antworten als Datei (Schluß)

Forth: Programmieren in der vierten Dimension

Eine Programmiersprache, die sich in der letzten Zeit ständig steigender Beliebheit erfreut, ist Forth. Zu Recht, denn Forth ist eine äußerst leistungsfähige Sprache mit einem ungewöhnlichen Konzept. Schwer zu lernen ist sie nicht.

Seit es Computer gibt, ist das zentrale Problem die Kommunikationsschnittstelle Mensch/ Computer - oder anders formuliert, wie sage ich meinem Computer, was er tun soll? Während in der »grauen Vorgeschichte« der EDV noch bitweise programmiert werden mußte, erwies sich dieses Verfahren im Laufe der Zeit als viel zu umständlich. Es entstanden höhere Programmiersprachen, die sich einer bestimmten Anzahl von Befehlsworten bedienen. Meist sind diese an die menschliche Sprache angelehnt. Zu solchen höheren Programmiersprachen zählen beispielsweise Fortran, Basic, Pascal, C oder Lisp, um nur einige zu nennen.

Betrachtet man sich die Entwicklung in der Computertechnik, so stellt man fest, daß sich, seitdem das erste Röhrengerät in Betrieb ging, bis zum heutigen Tage die Hardware rasant weiterentwickelt hat. Anders die Software. Hier dominieren noch immer Sprachen wie Fortran und Basic, die in ihren Ursprüngen in die fünfziger Jahre zurückreichen.

Daß Programmiersprachen für Computer, die Sie heute nur noch im Museum bewundern können, nicht immer den heutigen Ansprüchen genügen, ist leicht einzusehen. Vor allem Basic, das sich mittlerweile zum Standard für kleine und mittlere Systeme entwickelt hat, hinkt den Ansprüchen der meisten Programmentwickler weit hinterher. Zwar ist es leicht zu erlernen und besitzt eine unkomplizierte Befehlssyntax, aber strukturierte Programme, lokale Variablen und ähnliches sind für die meisten Dialekte tabu.

Was tut also der (Basic-)frustrierte Programmierer? Er sucht sich eine neue Sprache, die seinen Ansprüchen besser gerecht wird. Und da beginnt das Dilemma. Welche der vielen Konkurrenten kommt für ihn in Frage? Hier ist es hilfreich, sich zu überlegen, was die »neue Traumsprache« alles leisten soll. Folgende Punkte sind dabei für jeden Programmierer wichtig. Die Sprache soll:

- schnell sein,
- strukturiertes Programmieren ermöglichen,
- auf andere Computertypen übertragbar sein,
- die Stärken des eigenen Geräts unterstützen,
- erweiterbar sein,
- möglichst wenig des kostbaren RAM-Speichers belegen,
- relativ leicht zu erlernen und zu verstehen sein.

Wenn das auch Ihre Vorstellungen einer nahezu idealen Programmiersprache sind, dann brauchen Sie nicht länger zu suchen. Denn solch eine Traumsprache gibt es schon - Forth.

Zwar ist auch Forth nicht die Programmiersprache schlechthin, doch weist sie zumindest die oben aufgeführten Vorteile (und noch einige mehr) auf. Was es damit tatsächlich auf sich hat, darüber klären Sie die nächsten Seiten auf. Sie ersetzen zwar kein Handbuch von Forth (es können auf so wenig Seiten niemals alle Anweisungen erklärt werden), aber wir wollen versuchen, Ihnen ein Gefühl für den typischen Charakter von Forth zu geben. Vielleicht kommen Sie auf den Geschmack, sich mit dieser faszinierenden Sprache näher zu befassen.

Forth, Sprache für den Weltraum

Forth wurde Ende der sechziger Jahre entwickelt und ursprünglich zur Steuerung und Auswertung der Meßdaten einer Sternwarte eingesetzt. Schon damals stand die Zukunft von Forth im wahrsten Sinne des Wortes in den Sternen, denn die amerikanische Weltraumbehörde NASA wählte Forth als Sprache für zukünftige Satellitenprogramme. Doch bevor es dazu kommen sollte, blieb es bis Ende der siebziger Jahre sehr ruhig um diese Sprache. 1977 gründete eine nichtkommerzielle Gruppe von Programmierern die »Forth Interest Group« (FIG) und machte Forth damit einer breiteren Öffentlichkeit zugänglich. Es entstand als Standard FIG-Forth, aus dem sich zwei Jahre später der Forth 79-Standard entwickelte. Beide Versionen sind heute im Heim- und Personal Computerbereich verbreitet. Der Unterschied dieser Dialekte ist nur gering, so daß sich FIG-Forth-Programme leicht in den 79-Standard umsetzen lassen - und umgekehrt.

In unserer Einführung wollen wir im wesentlichen das ältere FIG-Forth benutzen. Etwaige Abweichungen zum 79-Standard bleiben aber auch nicht unerwähnt.

Mittlerweile wird für fast jedes Computersystem eine Forth-Version angeboten. Da die Sprache kaum Speicherplatz beansprucht, ist sie ideal für kleinere Systeme geeignet. Im allgemeinen benötigt Forth nicht mehr als 10 KByte RAM-Bereich. Je nachdem, welche Extras zusätzlich implementiert sind, kann sich dieser Bereich natürlich erhöhen.

Forth kommt im Prinzip ohne Massenspeicher aus, weshalb sich ein Diskettenlaufwerk erübrigt. Da beim Heimcomputer die Programme nicht im ROM vorliegen, braucht man aber unbedingt einen Kassettenrecorder. Dennoch, wie bei allen Sprachen ist auch unter Forth der Gebrauch eines Diskettenlaufwerks angenehmer als ein Datenrecorder. Auch kommen einige Stärken von Forth nur mit einem Laufwerk zum Tragen.

Nachdem Sie Ihre Forth-Version geladen und gestartet haben, erscheint, zusammen mit einer Mitteilung über die Herkunft des Systems, der gleiche Cursor, den Sie sicher schon von Basic her bestens kennen.

Zwei verschiedene Wege gibt es, um sich mit der neuen Sprache vertraut zu machen. Entweder Sie starten einen »Trockenkurs« und studieren das Handbuch in allen Einzelheiten, oder aber Sie legen alle Hemmungen ab, geben irgendetwas ein und warten, was der Computer machen wird. Entscheiden Sie sich für die zweite Art, so werden Sie sehr oft eine nüchterne Fehlermeldung auf dem Bildschirm vorfinden. Diese besteht entweder aus einem Kommentar wie zum Beispiel »CAN'T FIND« oder aus einer Ziffer, die die Fehlerart anzeigt. Manchmal wird der Computer aber auch mit einem lapidaren »OK« antworten. Immer dann haben Sie eine Forth-Anweisung richtig benutzt.

Für den Fall, daß Sie aber den Wort-



schatz ohne langes Ausprobieren kennenlernen wollen, ist in fast jedem Forth-Compiler eine Anweisung vorgesehen, die alle Befehle auf dem Bildschirm ausgibt. »VLIST« oder »WORDS« sind zwei häufig gebrauchte Kennworte für diesen Befehl. Dieses Wörterbuch zeigt sämtliche Anweisungen, die Ihr Compiler versteht. Ganz egal, ob es sich um vor- oder selbstdefinierte Befehle handelt.

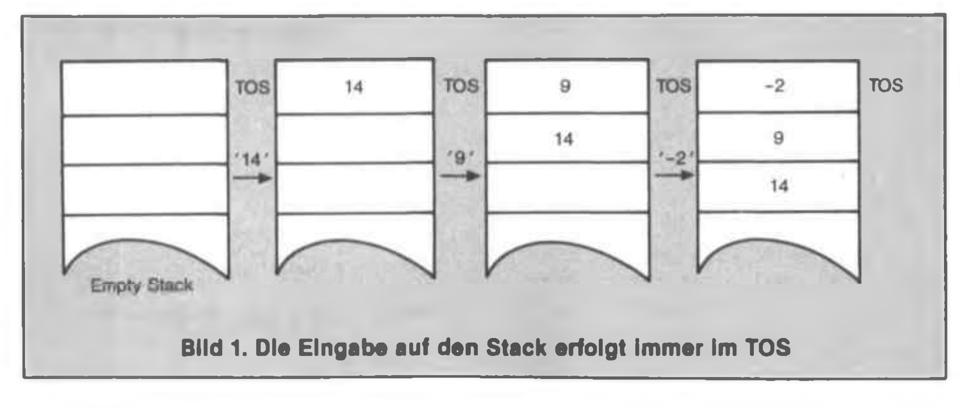


Bild 1. Die Eingabe auf den Stack erfolgt immer im TOS

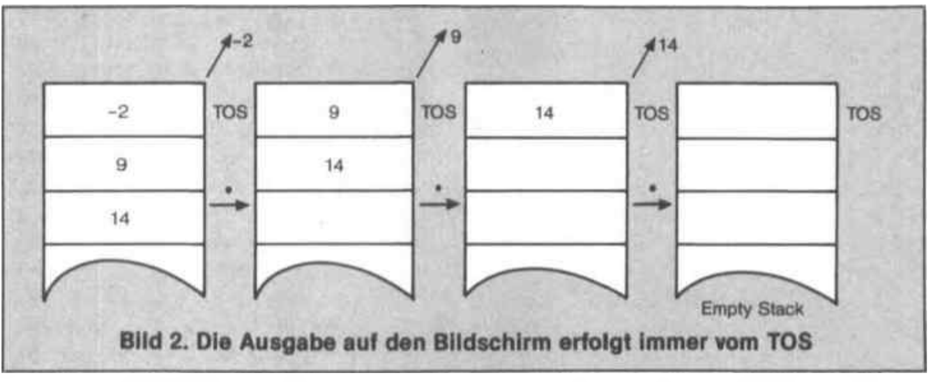


Bild 2. Die Ausgabe auf den Bildschirm erfolgt immer vom TOS

Erste Kontaktaufnahme

Unter Forth müssen alle Eingaben mit der RETURN- (oder ENTER-) Taste abgeschlossen werden. Die eingegebenen Worte lassen sich in drei Gruppen unterscheiden – in »Nonsens«-Worte, die der Compiler mit einer Fehlermeldung quittiert, in Anweisungen oder in Zahlen. Denn auch wenn Sie nur eine Zahl eingeben, reagiert der Compiler darauf mit »OK«, das heißt, er akzeptiert den Befehl ohne Probleme. Was ist nun mit dieser Zahl geschehen?

Bei nahezu allen Operationen in Forth spielt der Stack eine zentrale Rolle. Der Stack (zu deutsch Stapelspeicher) ist nichts anders als ein kleiner Speicherbereich, der nach einem besonderen Prinzip verwaltet wird. Jede Zahleneingabe von der Tastatur, die mit RETURN abschließt, landet zuerst einmal im »Top Of Stack« (TOS), also in der obersten Speicherzelle (eine Speicherzelle ist ein 16-Bit-Register) des Stacks.

Jede neu hinzukommende Zahleneingabe wird ebenfalls dort abgelegt. Alle bereits vorhandenen Zahlen rutschen um eine Position nach unten. Bei der Ausgabe vom Stack kommt als erstes die Zahl im TOS an die Reihe. Da es sich hierbei immer um die zuletzt eingegebene Zahl handelt, wird das Ganze als »Zuletzt rein – zuerst raus«-Prinzip bezeichnet. Im Fachenglisch heißt das dann »Last In – First Out«-Prinzip (kurz LIFO).

Alle Operationen, die den Stack beeinflussen, arbeiten nach diesem Verfahren. Deswegen sollten Sie sich damit gut vertraut machen. Bereitet Ihnen die Arbeitsweise noch Schwierigkeiten, dann hilft Ihnen vielleicht folgender Vergleich weiter.

Stellen Sie sich einen Schreibtisch vor, auf dem ein Stoß Papier liegt. Legen Sie ein Blatt auf dem Stoß ab, so landet es auf der obersten Stelle. Das nächste Blatt liegt darüber und an oberster Position befindet sich damit immer das Blatt, das zuletzt abgelegt wurde. Wollen Sie nun den Papierstoß abarbeiten, dann nehmen Sie zuerst das oberste Blatt weg. Und dieses ist das zuletzt hingelegte. Das schon am längsten auf dem Tisch liegende Papier ist das letzte beim Abarbeiten. Und so funktioniert auch der Stack in Ihrem Computer.

Nach soviel Theorie zurück zu Forth. Wir wissen nun, wie der Stack verwaltet wird und wollen ausprobieren, wie wir drei Zahlen auf den Stack legen und wieder herunterholen können. Wir geben einfach folgende Zeile ein:

```
14 9 -2
```

Zwischen zwei Zahlen muß immer ein Leerzeichen stehen und die Zeile mit der RETURN-Taste beendet werden. Der Computer quittiert die Eingabe mit »OK«. Was ist nun aber auf dem Stack passiert? Dazu betrachten wir Bild 1, das uns zuerst den »leeren« Stack und zum Schluß den »vollen« Stack zeigt. Zuerst wurde die 14 im TOS abgelegt, dann die 9 (gleichzeitig wandert die 14 um eins nach unten) und zum Schluß die -2 (die andern beiden Werte wandern nach unten).

Um diese Zahlen wieder vom Stack auszugeben, lernen Sie nun Ihr erstes Forth-Wort kennen (Befehle, bezie-

Forth-Steckbrief

- Forth wurde Ende der sechziger Jahre in den USA entwickelt.
- Forth ist eine Compilersprache, mit der Sie aber auch interaktiv arbeiten können.
- Forth ist sehr schnell. Vergleichbare Programme brauchen in Basic bis zu 20mal mehr Zeit als in Forth.
- Der Befehlssatz in Forth besteht aus 200 bis 300 Wörtern. Ein Wort läßt sich mit einem Unterprogramm in Basic oder besser mit einer Prozedur in Pascal vergleichen. Der Benutzer kann diesen Wortschatz um eigene Definitionen erweitern. Diese neuen Wörter sind im Gebrauch mit den Standard-Forth-Wörtern vollkommen identisch.
- Bei Forth-Wörtern handelt es sich entweder um Secondaries, die wiederum aus Forth-Wörtern aufgebaut sind, oder um Primitives, welche in Maschinencode definiert sind.
- Forth rechnet nur mit Integerzahlen von 16 oder 32 Bit Breite.
- Forth belegt in der Regel zwischen 8 und 12 KByte Speicherplatz (hängt von dem Umfang des Wortschatzes ab).
- Forth verwaltet den Diskettenspeicher virtuell, das heißt RAM- und Diskettenspeicher sind formell gleichwertig.
- Forth ist weitgehend standardisiert, das heißt alle für Heim- und Personal Computer angebotenen Versionen leiten sich vom FIG-Forth ab. FIG-Forth stammt von der Forth Interest Group, einer nichtkommerziellen Vereinigung von Programmierer.
- Die Rechenoperationen werden in Forth nach den Regeln der Umgekehrten Polnischen Notation (UPN) durchgeführt.

hungsweise Anweisungen, werden in Forth als Wort bezeichnet). Es handelt sich um einen unscheinbaren Punkt. Durch ».« wird die Zahl im TOS (Top Of Stack) auf den Bildschirm ausgegeben.

```
. -2 ok
```

Nach der Eingabe des Punktes und der RETURN-Taste reagiert der Computer mit der Meldung »-2 OK«. Wir wollen im folgenden immer die Zeile so angeben, wie Sie nach der Bearbeitung aussieht. Eingeben dürfen Sie natürlich nur die Forth-Wörter (in diesem Falle also nur den Punkt). Sie können natürlich auch mehrere Zahlen auf einmal ausgeben lassen:

```
. . 9 14 OK
```

Der nächste Punkt veranlaßt den Computer zu einer Fehlermeldung, da der Stack leer ist:

```
. 0 EMPTY STACK
```

Bild 2 zeigt Ihnen, wie sich der Stack bei der Ausgabe der Zahlen verändert. Sie sehen dabei, daß jede Zahl, die durch ».« auf dem Bildschirm erscheint, gleichzeitig vom Stack verschwindet und sich alle anderen Zahlen um eine Position nach oben bewegen.

Bevor wir zu unseren ersten Rechenaufgaben in Forth kommen, müssen wir uns noch einmal ganz genau mit der Syntax von Forth auseinandersetzen. Befehlswörter dürfen in Forth beliebig in einer Zeile stehen. Allerdings muß immer mindestens ein Leerzeichen zwei Anweisungen trennen. Anders als beispielsweise in Basic, wo es einen festgelegten, im Grunde nicht mehr erweiterbaren Befehlssatz gibt, kann jedermann Forth um neue Befehle bereichern. Da dabei jede Zeichenkombination als Wortname erlaubt ist, stellen die Leerzeichen für den Textinterpreter die einzige Möglichkeit dar, die Wörter voneinander zu unterscheiden.

. (n -)	-gibt die Zahl im TOS (oberste Zahl im Stack) aus
. »Text«	-gibt die Symbole zwischen den Anführungszeichen als Text aus
CR	-bewirkt einen Zeilenvorschub

Tabelle. Ihre ersten Wörter in Forth

Auch Texte lassen sich unter Forth auf den Bildschirm ausgeben. Dazu dient ein zweites Wort: »."«. Ein Beispiel:

```
." OSTERHASE" OSTERHASE OK
```

Wenn Sie ».″OSTERHASE″« mit RETURN an den Computer abschikken, dann gibt er den Text »OSTERHASE« zurück. Denken Sie an die richtige Verteilung der Leerzeichen, da sonst der Computer Sie nicht verstehen kann.

Mit solch einer Textausgabe können wir auch unsere Stack-Ausgabe komfortabler gestalten:

```
4 OK
CR ." TOS : " .
TOS : 4 OK
```

Wenn Sie die 4 und die zweite Zeile eingegeben haben, dann antwortet der Computer mit der dritten. Das neue Wort »CR« bewirkt einen Zeilenvorschub. Bei manchen Compilern funktioniert dieser letzte Befehl nur in Wörtern. Wie Sie solche definieren, erfahren Sie später. Manchmal darf ».″...″« durch ».(...)« ersetzt werden. Näheres finden Sie in Ihrem Handbuch. Die Wörter des ersten Teils der Einführung in Forth faßt die Tabelle noch einmal zusammen.

(Peter Monadjemi/hg)

UPN – Rechnen in der umgekehrten Polnischen Notation

Forth zeigt einige unkonventionelle Lösungswege, Computerprogramme zu erzeugen. Besonders das Rechnen in Forth unterscheidet sich von fast allen anderen Computersprachen.

Der Stack ist das wichtigste Hilfsmittel zum Rechnen in Forth. Dazu stehen verschiedene Operatoren und Befehlswörter zur Verfügung. Allerdings muß man bei dem Jonglieren mit Zahlen in Forth einige spezifische Besonderheiten beachten:
- Alle Operationen werden nach den Regeln der Umgekehrten Polnischen Notation (UPN) durchgeführt.
- Forth rechnet nur mit Integerzahlen (ganze Zahlen).
- Forth kennt in der Standard-Version nur die vier Grundrechenarten.

UPN ist eine Vorschrift für die Durchführung von Rechenoperationen unter Einbeziehung eines oder mehrerer Stacks. Nehmen Sie als Beispiel die Operation »33∗4=« (hier noch in der Basic-typischen Schreibweise). Die UPN-Schreibweise bereitet die Befehlszeile so auf, wie sie der Computer am leichtesten bearbeiten kann. Das bedeutet, daß zuerst die Operanden (Zahlen) und dann die Operatoren (Rechenzeichen) eingegeben werden. Unser Beispiel sieht dann wie folgt aus:

```
33 4 *
```

Das Ergebnis befindet sich nach der Berechnung im Stack – und zwar im TOS (Top of Stack) – und kann dort weiterverarbeitet werden. Der Vorteil der UPN gegenüber der Infix-Notation (das ist die Basic-übliche Eingabe von Berechnungen) macht sich erst bei größeren Ausdrücken bemerkbar. Die Eingabe in Infix-Schreibweise:

```
(2+7)/(4*(8-3))=
```

braucht bedeutend mehr Platz als die UPN-Schreibweise:

```
2 7 + 4 8 3 - * /
```

Sie sehen, daß die UPN weder Klammern noch Gleichheitszeichen benötigt. Dadurch ergibt sich neben einer kürzeren, Speicherplatz sparenden Schreibweise auch ein erheblicher Geschwindigkeitsvorteil. Die Klammern erfordern vom Computer nämlich immer ein Vorausschauen, »wo wird diese wieder geschlossen«. Bei der UPN hingegen werden bei jedem Rechenoperator die beiden obersten Werte im Stack miteinander verknüpft und deren Ergebnis direkt im TOS abgelegt. Für unsere Aufgabe bedeutet das, daß mit Eingabe des Pluszeichens die 2 und 7 addiert und das Ergebnis 9 in den TOS gelegt wird. Die Eingabe der drei Zahlen 4, 8 und 3 bewirkt, daß die 9 an der vierten Stelle von oben liegt. Das Minuszeichen berechnet 8 minus 3, legt die 5 in den TOS und zieht die beiden anderen

Werte nach oben. Das Malzeichen multipliziert die 5 mit der 4, legt das Ergebnis ab und schon steht die 9 direkt unter der 20. Das Divisionszeichen besorgt den Rest, so daß zum Schluß das Ergebnis der Rechnung im TOS steht. Die UPN ist also nicht etwa ein exotisches Rechenverfahren, sondern die »natürlichste« und effektivste Methode für einen Computer, Rechenoperationen zu verarbeiten.

Nun wollen wir uns aber damit befassen, wie sich der Stack verändert, wenn wir die Grundrechenarten bearbeiten lassen. Die Addition löst das Zeichen »+« aus.

```
4 5 + OK
```

Was ist nun auf dem Stack geschehen? Dazu betrachten wir uns Bild 1. Sie sehen, daß sich vor dem Aufruf von »+« die beiden zu addierenden Zahlen an den beiden obersten Stellen des Stacks befinden müssen. Nach der Addition wird das Ergebnis im TOS abgelegt. Von dort kann man es mit ».« leicht auf den Bildschirm ausgeben.

```
. 9 OK
```

Die 9 zeigt an, daß das Ergebnis tatsächlich im TOS gespeichert war. Die Subtraktion erfolgt analog:

```
16 12 - . 4 OK
```

Auch hier müssen sich die beiden Zahlen zuerst auf dem Stack befinden. »-« zieht die Zahl im TOS von der darunterliegenden ab (siehe Bild 2) und »/« ruft die Division auf (Bild 3):

```
20 2 / . 10 OK
```

Bis dahin ist die Welt noch in Ordnung. Doch das nächste Beispiel führt zu einem unerwarteten Ergebnis:

```
21 2 / . 10 OK
```

Hier kommt das schon erwähnte Fehlen von Real-(Fließkomma-)zahlen voll zum Tragen. Aber keine Bange: Forth bietet verschiedene Wege, auch beliebig genaue Ergebnisse zu erhalten. Mit dem Operator »/« bekommen Sie also nur die Vorkommazahl. Um den ganzzahligen Rest dieser Rechenoperation zu erhalten, gibt es unter Forth einen Befehl, den die meisten Basic-Dialekte nicht kennen.

```
21 2 MOD . 1 OK
23 4 MOD . 3 OK
```

21 geteilt durch 2 gibt 10, Rest 1 und 23 geteilt durch 4 gibt 5, Rest 3. Die ganzzahligen Restbeträge werden bei diesem Wort in den TOS gelegt.

Die Multiplikation aktiviert »*«.

```
20 20 * . 400 OK
```

Allerdings muß man auch hier aufpassen, nicht den erlaubten Bereich zu verlassen. So ergibt

```
200 200 * . -25534 OK
```



ein falsches Ergebnis (beziehungsweise eine Fehlermeldung). Diesmal liegt es allerdings nicht an den Integerzahlen, sondern an der Tatsache, daß Forth intern nur mit 16 Bit breiten Zahlen rechnet. Da das 16. Bit das Vorzeichen enthält, ist damit der Rechenbereich auf die Zahlen zwischen -32768 und +32767 beschränkt -, zugegebenermaßen ein kleiner Darstellungsbereich. Doch auch hier stehen dem Programmierer alle Türen offen. Theoretisch können Sie mit beliebig breiten Zahlen arbeiten. Von Haus aus erlaubt Forth, entweder auf das Vorzeichen zu verzichten oder aber mit 32 Bit breiten Zahlen zu arbeiten. Beide Fälle setzen allerdings spezielle Wörter voraus. So erhalten Sie bei unserer »verunglückten« Multiplikation mit

```
200 200 * U. 40000 OK
```

doch noch ein vernünftiges Ergebnis. Mit »U.« wird die Zahl, die im TOS steht, als vorzeichenlose Zahl auf dem Bildschirm ausgegeben.

Das Rechnen mit 32-Bit-Zahlen erfordert spezielle Wörter. Diese erkennt man fast immer an einem »D« oder einer »2«, mit der der eigentliche Befehl beginnt. Damit der Textinterpreter solch eine doppelt lange Zahl korrekt erkennt, muß diese mit einem Dezimalpunkt eingegeben werden. Allerdings spielt es dabei keine Rolle, an welcher Stelle dieser Punkt steht.

```
222.222 OK
44.4444 OK
D+ OK
```

Durch »D+« werden die beiden doppelt langen Zahlen im Stack addiert und das Ergebnis im TOS abgelegt. Wie bekommen Sie nun diesen Wert auf den Bildschirm? Sicher nicht mit ».«. Denn damit erhalten Sie nur die niederwertigen 16 Bit der 32-Bit-Zahl ausgegeben. Auch hier ist ein besonderes Ausgabewort notwendig, nämlich »D.«.

```
D. 666666 OK
```

Bislang wurde die Anordung der Zahlen auf dem Stack durch die Reihenfolge der Eingabe festgelegt. Sehr oft besteht jedoch die Notwendigkeit, diese Reihenfolge zu verändern, beziehungsweise einzelne Zahlen auf dem Stack zu kopieren. Auch dazu stehen in Forth eine Reihe von Befehlen zur Verfügung. Die wichtigsten erklären wir Ihnen im folgenden.

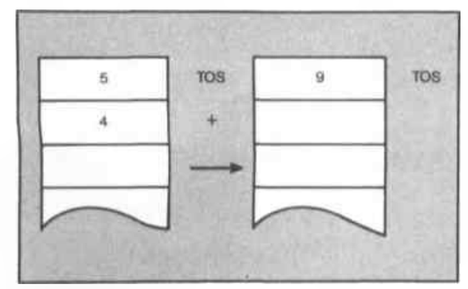


Bild 1. So verändert sich der Stack bei der Addition

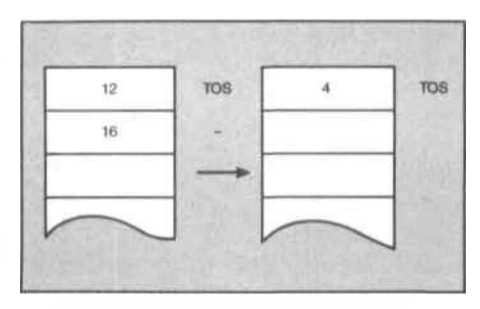


Bild 2. So verändert sich der Stack bei der Subtraktion

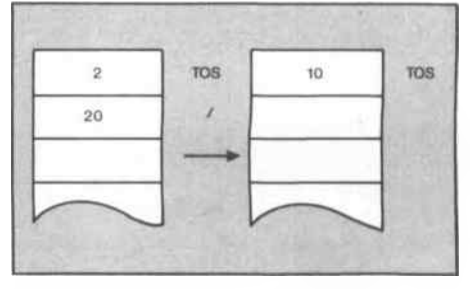


Bild 3. So verändert sich der Stack bei der Division

```
34 DUP OK
. . 34 34 OK
```

»DUP« (Bild 4) kopiert die Zahl im TOS und verschiebt die darunterliegenden um eins nach unten. Die Operation ist beispielsweise immer dann notwendig, wenn eine Zahl im TOS zwar ausgegeben, mit ihr aber noch weiter gerechnet werden soll.

```
67 88 SWAP OK
. . 67 88 OK
```

»SWAP« (Bild 5) vertauscht die beiden obersten Zahlen im Stack. Normalerweise muß ja die zuletzt eingegebene Zahl als erste wieder auf dem Bildschirm erscheinen (LIFO-Prinzip: Last in first out). Das wäre hier die 88 gewesen. »SWAP« hat nun aber die beiden obersten Werte im Stack vertauscht, so daß die 67 im TOS stand und damit auch zuerst ausgegeben wurde.

```
12 33 65 ROT OK
 . . . 12 65 33 OK
```

»ROT« läßt die obersten drei Zahlen im Stack einmal gegen den Uhrzeigersinn rotieren. Wie sich dabei der Stack verändert, zeigt am besten Bild 6. Durch »ROT« wird die Zahl von der dritten Stelle im Stack ins TOS gebracht, während die beiden darüberliegenden Werte um eine Position nach unten wandern. Alle Befehle (und noch einige mehr) finden Sie in der Tabelle noch einmal zusammengefaßt.

Normalerweise werden alle Ein- und Ausgaben von Zahlen im Dezimalsystem durchgeführt. Forth ist jedoch in der Lage, beispielsweise die Zahlen in nahezu jedem System auszugeben. Dazu ist lediglich der Inhalt einer einzigen User-Variablen mit dem Namen »BASE« zu ändern. Bei den User-Variablen handelt es sich um Speicher-

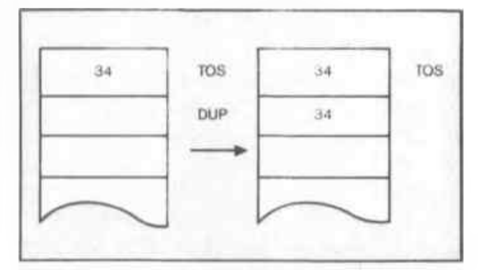


Bild 4. »DUP« verdoppelt die Zahl im TOS

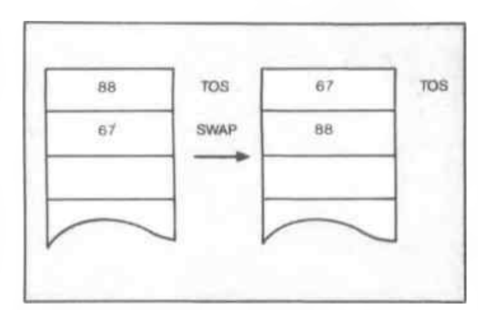


Bild 5. »SWAP« vertauscht den Wert im TOS mit dem darunterliegenden Stackwert

Die Erklärung der Darstellung der Befehle finden Sie im letzten Kapitel.

Befehl	Beschreibung
+ (n1 n2 bis n3)	- addiert die beiden obersten Zahlen des Stacks und legt das Ergebnis im TOS ab
- (n1 n2 bis n3)	- subtrahiert n2 von n1 und legt das Ergebnis im TOS ab
/ (n1 n2 bis n3)	- dividiert n1 durch n2 und legt das Ergebnis im TOS ab
MOD (n1 n2 bis n3)	- dividiert n1 durch n2 und legt den ganzzahligen Rest im TOS ab
c@ (a bis n)	- holt ein Byte und speichert es im TOS
? (a bis n)*	- wie »c@«, aber mit Ausgabe auf dem Bildschirm
! (n a usw.)	- speichert Zahl in der Adresse a, die im TOS angegeben ist

Tabelle. Die in diesem Abschnitt neu vorgestellten Befehle und deren »Geschwister«

werte, die wichtige Systemgrößen beinhalten. So enthält zum Beispiel »SO« die Adresse des Stacks, »DP« den Beginn des Wörterbuches und »BASE« den Wert der aktuellen Zahlenbasis.

Der Aufruf einer User-Variablen holt jedoch in Forth nicht den Wert, sondern die Speicheradresse, unter der der Wert zu finden ist. Um den aktuellen Wert von »BASE« zu erfahren, brauchen wir ein Wort (Anweisung) vom Typ: »Gib den Inhalt der Speicherzelle mit der Adresse addr aus«. Solch ein Wort stellt »c@« (manchmal auch »?«) zur Verfügung.

```
BASE C . 10 OK
```

Diese Anweisung bringt den aktuellen Wert der User-Variablen BASE auf den Bildschirm. Um den Wert zu ändern, benötigen wir ein Wort vom Typ: »Lege den Wert a unter der Adresse addr ab«. Dieses Wort lautet »!«. Sowohl der Wert »a« als auch die Adresse »addr« müssen sich auf dem Stack befinden – und zwar in der richtigen Reihenfolge. Nach der Eingabe muß die Adresse (in diesem Fall BASE) im TOS stehen.

```
63 2 BASE ! OK
 . 111111 OK
```

Die Zahlen 63 und 2 werden auf den Stack gelegt, dann die Adresse BASE im TOS gespeichert. Das Wort »!« ändert die Ausgabefunktion auf Dualzahlen. Mit ».« erscheint dann die 63 in dualer Form (111111) auf dem Bildschirm. Da wir gerade auf Dualzahlen umgeschaltet haben, muß jetzt auch die Eingabe in dualer Form erfolgen.

```
9 23 ? CAN'T FIND
```

Die Zahlen 9 und 23 sind keine Dualzahlen und deshalb gibt der Compiler eine Fehlermeldung (hier »CAN'T FIND«) zurück. Mit

```
11 BASE ! OK
```

schalten wir auf die Zahlenbasis »3« um (11 dual ist 3 dez). Da es nun etwas kompliziert ist, die richtige Ziffernfolge für Dezimalzahlen zu finden (10 dez = 1010 dual = 101 zur Basis 3), gibt es unter Forth das Wort »DECIMAL«, das immer wieder zum vertrauten Dezimalsystem zurückführt.

```
DECIMAL OK
```

Ein kleines Beispiel zeigt eine beeindruckende Lösung eines Problems, das in vielen anderen Sprachen nur bedeutend umständlicher gelöst werden kann.

```
: DUAL 2 BASE ! ; OK
: DOPPEL
  20 0 DO CR I DECIMAL .
       I DUAL . LOOP ; OK
```

Wir haben zuerst das Wort »DUAL« zur Umschaltung auf Dualzahlen definiert und dann die Anweisung »DOPPEL«. Wie man Wörter bestimmt und auch wie die Schleife »DO ... LOOP« arbeitet, finden Sie in den folgenden Kapiteln. Hier sollen Sie das Programm nur eintippen und starten. Nach dem Aufruf von »DOPPEL« ergibt sich auf dem Bildschirm folgendes Bild:

```
0 0
1 1
2 10
3 11
4 100
5 101
6 110
...
```

(Peter Monadjemi/hg)

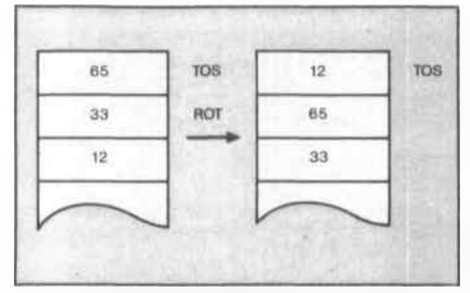


Bild 6. »ROT« bringt den dritthöchsten Wert in den TOS und schiebt die beiden anderen Werte um eins weiter nach unten

Forth lernt dazu

Einen der größten Vorteile von Forth macht der sehr einfache Ausbau seines Wortschatzes aus. Jede Forth-Version kann sein Benutzer mit neuen, selbstdefinierten Anweisungen erweitern.

Anders als beispielsweise in Basic bestehen Forth-Programme nicht aus einer bestimmten Anzahl von Programmzeilen, sondern aus Wörtern. Jedes diese Wörter kann weitere beinhalten, die wiederum neue Wörter enthalten dürfen und so weiter. Dieser extrem modulare Aufbau führt dazu, daß das eigentliche Hauptprogramm letzten Endes aus nur einem einzigen Wort bestehen kann.

Ein solches »Forth-Wort« läßt sich mit einem Unterprogramm in Basic oder einer Prozedur in Pascal vergleichen. Es gibt verschiedene Wege, sich selbst solch eine Anweisung zu definieren. Die einfachste ist die sogenannte »Colondefinition«, bei der das Wort weitere Wörter enthalten darf, die nach Aufruf des neuen Worts ausgeführt werden. Handelt es sich bei diesen Wörtern wiederum um Forth-Wörter, dann spricht man von SECUNDARIEs. Ein Wort, das direkt Maschinencode-Routinen aufruft, bezeichnet man als PRIMITIVE. Die meisten Wörter des Grundwortschatzes von Forth sind SECUNDARIEs.

Wie kann man nun solch ein Wort selbst definieren?

Jede Wortdefinition eines SECUNDARIEs leitet ein unscheinbarer Doppelpunkt »:« ein. Er bewirkt unter anderem, daß das System in den »Compile mode« umschaltet. Das hat zur Folge, daß alle nun folgenden Anweisungen nicht mehr direkt ausgeführt, sondern in das Wörterbuch eingetragen werden.

Unter einem Wörterbuch (englisch Dictionary) wird in Forth ein Speicherbereich verstanden, der den Wortschatz der betreffenden Version beinhaltet. Jede neue Wortdefinition wird nun auch in diesem Wörterbuch verzeichnet. Das Dictionary selbst ist in sogenannte Vokabulare unterteilt. Zwischen verschiedenen Vokabelbereichen schaltet VOKABULARY NAME um.

Durch Aufruf von NAME wird der Vokabelbereich NAME zum CONTEXT-VOKABULAR, das heißt dem aktuellen Vokabular, in dem die Dictionary-Suchläufe zuerst beginnen. Normalerweise ist das Standard-Vokabulary eingeschaltet. Es trägt den Namen Forth.

Wenn wir jetzt ein neues Wort definieren, so wird dies in das »Haupt«-Wörterbuch eingetragen. Durch »WORDS«, »VLIST« oder einem anderen, compilerspezifischen Namen kann man das überprüfen, denn dieser gibt ja den gesamten Inhalt des Wörterbuchs aus. Das Wörterbuch baut sich übrigens in Richtung größer werdender Adressen auf. Das Ende, also den Beginn des freien Arbeitsspeichers, kann man mit »HERE« ins »Top of Stacks« (TOS) laden. Nun aber zurück zu unserem Problem, selbst neue Worte zu definieren.

Anders als in jeder mittelmäßigen Basic-Version fehlt in Forth die Quadratfunktion. Sie ist aber relativ einfach zu definieren. Die Zahl, welche quadriert werden soll, muß zuerst einmal im TOS stehen. Dann kopieren wir sie mit DUP und multiplizieren beide miteinander. Die beiden Wörter, die wir dazu brauchen, kennen Sie schon.

```
4 DUP * . 16 OK
16 DUP * . 256 OK
```

Unsere Überlegung scheint zu stimmen, denn in beiden Fällen wurde die Ausgangszahl quadriert.

Um die Quadratfunktion nun im Wörterbuch zu »verewigen«, erweitern wir es um das Wort »QUADRAT«. Diese Funktion soll bei ihrem Aufruf immer den aktuellen Wert im TOS quadrieren und das Ergebnis dort auch wieder ablegen.

Wie schon erwähnt, leitet ein Doppelpunkt die neue Wortdefinition ein. Danach folgt der Name der neuen Funktion, dann die Befehlsfolge. Ein Semikolon schließt das Ganze ab. Mit

```
: QUADRAT DUP * ; OK
```

haben Sie nun den Wortschatz Ihres Systems bereichert. Bevor Sie das neue Wort aufrufen, müssen Sie aber daran denken, daß die Zahl, die quadriert werden soll, im TOS steht.

```
12 QUADRAT OK
```

berechnet das Quadrat von 12. Das Ergebnis 144 bekommen wir mit

```
. 144 OK
```

auf den Bildschirm. An oberster Stelle im Wörterbuch steht nun das neue Wort. Falls Ihnen die Ausgabeform zu nüchtern ist, dann definieren Sie doch mit

```
: AUSGABE CR
 ." DIE QUADRATZAHL IST" . ; OK
```

eine Ausgaberoutine. Nach dem Aufruf von AUSGABE erscheint die gerade aktuelle Zahl aus dem TOS. Der Bildschirm sieht dann wie folgt aus:

```
12 QUADRAT AUSGABE
DIE QUADRATZAHL IST 144
```

Um nun die Ausgaberoutine in dem Wort QUADRAT gleich mit aufzurufen, bedarf es einer Neudefinition dieses Wortes. Es gibt zwar auch Wege, bestehende Routinen zu verändern, aber das lassen wir hier beiseite. Also geben wir das Wort schnell noch einmal neu ein.

```
: QUADRAT DUP * AUSGABE ;
? QUADRAT ISN'T UNIQUE
```

Lassen Sie sich durch die Fehlermeldung nicht irritieren. Damit teilt Ihnen der Compiler nur mit, daß es bereits ein Wort mit diesem Namen gab. Durch die neue Definition haben Sie es aber umbenannt. Rufen Sie jetzt QUADRAT auf, und Sie erhalten das gewünschte Ergebnis:

```
5 QUADRAT
DIE QUADRATZAHL IST 25 OK
```

Ein erneuter Blick ins Wörterbuch zeigt, daß jetzt zwei Wörter mit dem Namen QUADRAT existieren. Um ein Wort zu löschen, benutzt man FORGET.

```
FORGET QUADRAT OK
```

Damit bleibt nur noch die Frage zu klären, welche Version der beiden Worte QUADRAT gelöscht wurde. Am einfachsten ist das festzustellen, indem man QUADRAT noch einmal aufruft.

```
10 QUADRAT OK
```

Damit ist klar, daß FORGET die neueste Version unseres Wortes gelöscht hat. Doch damit nicht genug. FORGET löscht nicht nur das betreffende Wort, sondern auch alle anderen, die später definiert wurden. Deshalb sollte man FORGET nur sehr vorsichtig einsetzen.

Der Grund für diese Wirkungsweise von FORGET ist leicht zu verstehen, wenn man sich vorstellt, daß alle Wörter im Wörterbuch durch eine »KETTE« miteinander verbunden sind. Trennen Sie die Kette an einer Stelle (etwa durch FORGET), so sind auch alle Wörter verloren, die bis zu diesem Punkt auf der Kette aufgereiht wurden.

Nicht immer läßt Forth das Löschen von Wörtern so ohne weiteres zu. In den meisten Forth-Versionen ist der Sprachkern geschützt. Eine User-Variable »FENCE« enthält die Adresse, ab der ein Löschen durch FORGET nicht mehr möglich ist.

Noch ein Wort zur Namensgebung. Hier dürfen Sie Ihrer Kreativität freien Lauf lassen, denn als Wortname ist jede beliebige Zeichenkombination erlaubt, die nicht länger als 31 Zeichen ist. Lediglich Leerzeichen dürfen nicht benutzt werden.

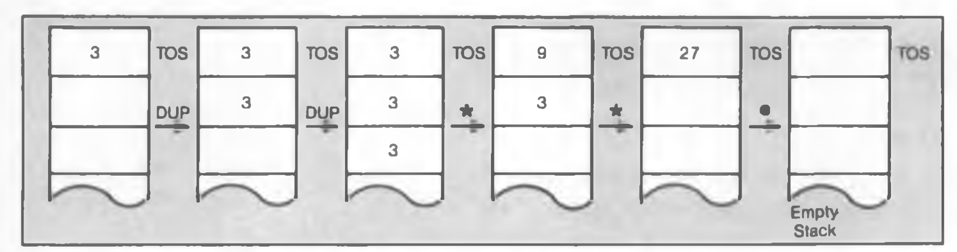
Daß sich in Forth so ziemlich alles um, beziehungsweise neu definieren läßt, zeigt das folgende Beispiel:

```
: 8 6 ; OK
```

Damit wurde der Zahl 8 kurzerhand eine neue »Bedeutung« gegeben. Denn auf einmal erhalten Sie mit

```
8 QUADRAT . 36 OK
```

ein recht merkwürdiges Ergebnis. Forth



So wird der Stack durch das Wort KUBIK verändert

: – Leitet die Definition eines Forth-Wortes ein.
; – Beendet die Definition eines Forth-Wortes.
FORGET – Löscht alle Wörter bis einschließlich dem angegebenen aus dem Wörterbuch.

Die neuen Worte dieses Abschnitts

ist, und das werden Sie noch öfters feststellen, die Sprache der nahezu unbegrenzten Möglichkeiten.

Um Ihnen die Wortbildung noch einmal zu verdeutlichen, definieren wir noch eine zweite Funktion.

```
: KUBIK DUP DUP * * . ;
```

Mit der Funktion KUBIK wird ab sofort die Kubikzahl des Ausgangswerts im TOS auf den Bildschirm ausgegeben. Dazu verdoppelt DUP die Zahl auf dem TOS zweimal und multipliziert dann die beiden obersten Werte jeweils miteinander (»* *«). Der Doppelpunkt und das Semikolon umschließen die Definition.

Mit

```
3 KUBIK 27 OK
```

bekommen wir das gesuchte Ergebnis.

```
100 KUBIK 16960 OK
```

ist jedoch ein etwas seltsames Ergebnis. Hier müssen wir uns wieder den Wertebereich unserer Zahlen ins Gedächtnis zurückrufen. Denn 1 000 000 können wir mit unserem Zahlenbereich zwischen -32768 und 32767 nicht darstellen.

Nachdem Sie nun eine ganze Menge über die Wortdefinition und das Forth-Wörterbuch gelernt haben, ist es an der Zeit, eine Zusammenfassung durchzuführen:

– Jede Wortdefinition wird in das Wörterbuch eingetragen. Dabei ist der hier besprochene Doppelpunkt nicht der einzige Weg, eine Wortdefinition vorzunehmen.

– Ein Eintrag in das Wörterbuch hat zur Folge, daß das jeweilige Wort Bestandteil des Wortschatzes wird, und somit auch, genauso wie die Worte aus dem Grundwortschatz, aufgerufen werden kann.

– Durch FORGET NAME wird die letzte Definition NAME gelöscht, sowie alle danach durchgeführten Definitionen.

(Peter Monadjemi/hg)

Forth, entscheiden Sie sich!

UPN, Rechnen mit dem Stack und Wortdefinitionen sind nach den letzten Seiten kein Problem mehr für Sie. Aber ein Computer-Programm muß auch Entscheidungen treffen können.

Wie in fast allen höheren Programmiersprachen, können auch in Forth Entscheidungen in der Form »Führe eine Anweisung nur dann aus, wenn ein Vergleich positiv ausfällt« bearbeitet werden. Forth stellt dazu zwei Konstruktionen zur Verfügung. Zum einen »IF ... ENDIF« und zum andern »IF ... ELSE ... ENDIF«. Die zweite Anweisung bearbeitet den Teil zwischen IF und ELSE, wenn der Vergleich positiv ist. Ist er negativ, dann wird der Teil ausgeführt, der zwischen ELSE und ENDIF steht. Fast alle Forth-Dialekte erlauben anstelle von ENDIF auch THEN, wenn auch der erste Weg die sinnvollere Bezeichnung darstellt.

Ein Beispiel verdeutlicht die Arbeitsweise von IF ... ENDIF:

```
: TEST 9 >IF ."ZU GROSS !"
  ENDIF ; OK
```

»TEST« prüft, ob die Zahl im TOS (also die zuletzt eingegebene Zahl) größer als 9 ist. In diesem Fall wird der Kommentar »ZU GROSS !« ausgegeben.

```
4 TEST OK
11 TEST ZU GROSS OK
```

Wenn Sie sich nun einmal das zugehörige Stack-Diagramm (Bild 1) anschauen, dann wird Ihnen der Mechanismus der IF-ENDIF-Anweisung schnell klar. Um einen Vergleich durchzuführen, müssen sich zunächst einmal beide Zahlen im Stack befinden. Der Vergleichsoperator »>« holt beide Zahlen vom Stack, führt den Vergleich aus und legt für den Fall, daß er positiv ausfällt eine »1« und für den Fall, das er negativ ausfällt eine »0« im TOS ab. Von diesem Flag (deutsch: Flagge oder Signal) hängt es ab, ob die zwischen IF und ENDIF stehenden Anweisungen ausgeführt werden oder nicht.

Bei der IF-ELSE-ENDIF-Anweisung dagegen werden für den Fall, daß der Vergleich negativ ausfällt, die Anweisungen zwischen ELSE und ENDIF ausgeführt. Alle erlaubten Vergleichsoperatoren zeigt Tabelle 1.

Wenn Sie die Vergleichsoperatoren durchgehen, dann werden Sie sicher den Vergleich »ungleich« vermissen. Er läßt sich aber leicht durch die Wortfolge »= NOT« ersetzen. Das Flag, das bei »=« im TOS abgelegt wird, invertiert dann »NOT« und wir haben unser Ziel erreicht.

Zwei Punkte müssen Sie aber noch bedenken, wenn die Anweisung für eine Entscheidung dienen soll. Sie darf nie im Direktmodus, sondern nur innerhalb einer Wortdefinition stehen. Außerdem ist noch zu beachten, daß die Zahlen, mit denen man den Vergleich durchgeführt hat, anschließend vom Stack verschwunden sind. Wollen Sie weiter mit diesen Werten arbeiten, dann müssen Sie sie zuvor kopieren.

Nun zu einem anderen Thema: Die Stärke eines Computers liegt in seiner Fähigkeit, bestimmte Anweisungen beliebig oft sehr schnell zu wiederholen. Während im normalen Basic hierfür nur die FOR-NEXT-Schleife vorhanden ist, kennt Forth insgesamt vier verschiedene Anweisungen. Allen gemeinsam ist, daß sie nur in Wörtern (also nicht direkt) benutzt werden dürfen. Die einfachste Wiederholfunktion ist »DO LOOP«:

```
: SCHLEIFE 10 0 DO I .
          LOOP ; OK
```

Als Ergebnis bekommen wir

```
SCHLEIFE 0 1 2 3 4 5 6 7 8 9 OK
```

Zwei Besonderheiten fallen an der Schleifenkonstruktion sofort auf:

– Zuerst wird der End- und dann der Anfangswert übergeben.

– Das Wort »I« holt den Schleifenwert, der die bisherige Anzahl der Durchläufe angibt, in den TOS.

Die Bedeutung des Wortes »DO« besteht darin, zum einen die Stelle zu markieren, zu der nach »LOOP« unter Umständen zurückgekehrt wird, und dient zum anderen dazu, die beiden Schleifenwerte (Start und Ende) auf einen weiteren Stack zu transferieren. Von diesem war bisher noch nicht die Rede, da er für den Anfänger keine praktische Bedeutung hat. Die Aufgabe dieses »RETURN STACK« besteht

darin, wichtige Adressen bei der Ausführung eines Wortes zu verwalten. Und auch die Schleifenwerte einer DO-LOOP-Anweisung werden hier gespeichert. Damit ist auch die Bedeutung des Wortes »I« zu verstehen. Diese Anweisung holt den momentanen Wert des »Top Of Return Stack« in den TOS.

Die DO-LOOP-Anweisung gehört zu den sogenannten »definierten Schleifen«, da die Anzahl der Durchläufe von vornherein fest steht. Ganz anders ist das bei »BEGIN ... UNTIL«:

```
: TEST BEGIN 1 + DUP .
       DUP 100 =
       UNTIL ." FERTIG !" ; OK
```

Hier steht die Anzahl der Durchläufe nicht von vornherein fest. Sie hängt vielmehr von einer Bedingung ab. In unserem Beispiel wird durch »=« geprüft, ob der Inhalt des TOS bereits den Wert 100 erreicht hat. Ist das der Fall, so wird im TOS eine 1 als Flag abgelegt. Daran erkennt das Wort UNTIL, daß eine Anweisung abzubrechen ist.

Für den Fall, daß der Inhalt des TOS 100 noch nicht erreicht hat, wird durch »=« eine 0 im TOS abgelegt und anschließend werden alle Anweisungen, die zwischen BEGIN und UNTIL liegen, ein weiteres Mal wiederholt. Somit handelt es sich bei dieser Anweisung um eine vom Typ: »Wiederhole so lange, bis eine bestimmte Bedingung wahr ist.«

Falls Ihnen dieser Befehl immer noch zu undurchsichtig ist, so nehmen Sie ein Blatt Papier und zeichnen Sie die Stackbelegung bei dem Wort TEST auf. Denken Sie daran, daß sowohl durch ».«, als auch durch »=« der Inhalt des TOS vom Stack verschwindet, wenn man ihn nicht vorher kopiert hat.

Noch ein Beispiel für diese Befehlsfolge:

```
: UEBUNG BEGIN CR
  ." DRUECKE EINE TASTE" KEY
  65 = UNTIL ; OK
```

Durch KEY wird ein Zeichen von der Tastatur gelesen (ähnlich GET in Basic) und der dazugehörige ASCII-Wert im TOS abgelegt. Dieser wird daraufhin mit 65 verglichen (ASCII-Wert von A ist 65). Haben Sie tatsächlich A eingegeben, so bricht der Programmlauf ab, andernfalls wird er ein weiteres Mal durchgeführt.

Ähnlich wie bei BEGIN-UNTIL liegen die Verhältnisse bei der BEGIN-WHILE-REPEAT-Anweisung mit dem Unterschied, daß die Ausführungs-Bedingung bereits vor dem WHILE stehen muß. Die eigentliche Anweisung befindet sich zwischen WHILE und REPEAT. Ist die Bedingung nicht erfüllt, so wird die Anweisung gar nicht erst ausgeführt.

```
: TASTE BEGIN KEY 65 = 0=
 WHILE ." VERSUCH'S NOCHMAL "
 UNTIL ." NA ENDLICH !" ; OK
TASTE B VERSUCH'S NOCHMAL OK
TASTE A NA ENDLICH ! OK
```

KEY bringt wieder den ASCII-Code der gerade gedrückten Taste in den TOS. »65 =« prüft diesen Wert auf Code 65 (für A). Da in diesem Fall im TOS eine »1« abgelegt und dadurch die Anweisung zwischen WHILE und UNTIL nicht ausgeführt werden würde, wird der Inhalt des TOS (das Flag) mit »0=« invertiert. Wenn im TOS eine 0 liegt, dann verändert »0=« diese in eine 1.

Zum Abschluß dieses Kapitels noch eine Wiederholungsanweisung, bei der Sie sich keine Gedanken um ein Abbruchkriterium machen müssen. Denn bei »BEGIN-AGAIN« handelt es sich um eine Endlosanweisung.

```
: NONSTOP BEGIN 40 EMIT
          AGAIN ; OK
```

Nach dem Aufruf von NONSTOP produziert Ihr Computer so lange Klammeraffen, bis Sie den Strom abschalten oder einen Reset durchführen.

Variablen, ohne die kaum ein Basic- oder Pascal-Programm auskommt, haben wir bisher eher beiläufig erwähnt. Das liegt daran, daß Forth (anders als Basic und Pascal) stackorientiert ist. Damit ist gemeint, daß sich alle Zahlenoperationen auf dem Stack abspielen. Demnach könnte man streng genommen auf die Variablen völlig verzichten. In bestimmten Fällen haben diese aber doch ihre Bedeutung. Denn zum einen ist die Kapazität des Stacks begrenzt, und zum anderen ersparen Sie sich durch die Verwendung von Variablen, beziehungsweise Konstanten, allzu umständliche Stack-Operationen.

Um mit Konstanten oder Variablen zu arbeiten, müssen Sie diese (ähnlich Pascal) zuerst einmal definieren und ihnen einen Anfangswert zuweisen. Dies geschieht durch die Definitionswörter »CONSTANT« und »VARIABLE«.

```
0 VARIABLE ZAHL OK
```

Damit haben wir eine Variable auf den Namen Zahl definiert. Beim Aufruf von Zahl erhalten Sie nun aber nicht deren Wert, sondern lediglich die Adresse, unter der dieser Wert im Speicher zu finden ist.

```
ZAHL . 12345 OK
```

Um den eigentlichen Wert zu erfahren, benutzen wir den schon bekannten Befehl »C@« beziehungsweise »?«. Seine Bedeutung war: »Hole den Inhalt der Speicherzelle, deren Adresse im TOS liegt.« Mit

```
4 ZAHL ! OK
```

weisen wir unserer Variablen ZAHL den Wert 4 zu. Zuerst laden wir die 4 und die Adresse von ZAHL auf den Stack. Mit

```
ZAHL   4 OK
```

testen wir, ob die neue Zahl im Speicher abgelegt wurde.

Ein wenig anders schaut es mit den Konstanten aus. Hier bringt der Aufruf der Konstanten mit Namen ihren Wert direkt in den TOS:

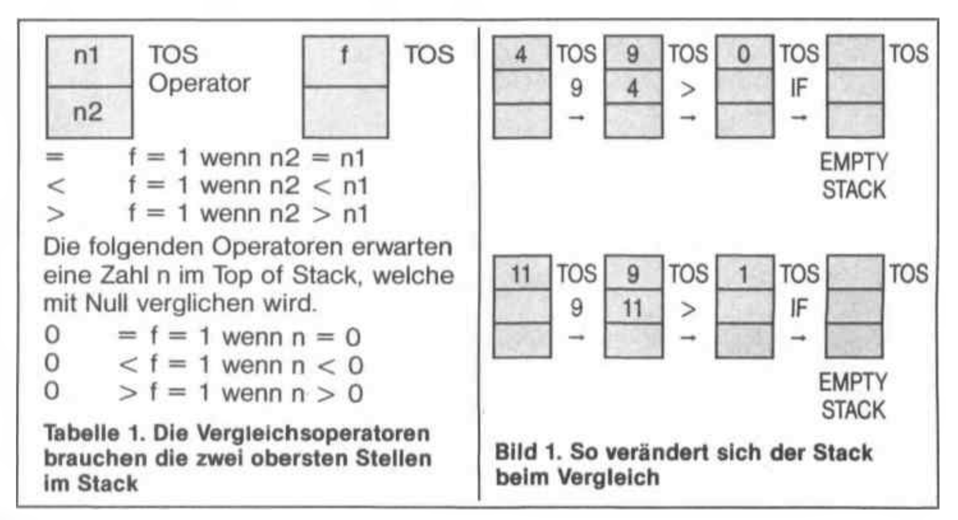
```
45 CONSTANT WERT OK
WERT . 45 OK
```

Auch der Wert einer Konstanten läßt sich ändern, wenn Sie die Adresse kennen. Als wir weiter vorne die Zahlenbasis veränderten, griffen wir auf solch eine Variable zurück. Zum Abschluß noch ein kleines Zählprogramm, das im Hexadezimalsystem bis 100 zählt:

```
: HZAEHL
  100 0 DO HEX I . CR LOOP
  DECIMAL ; OK
```

Programmieren können Sie in Forth jetzt natürlich noch lange nicht. Aber Sie kennen die Grundzüge und es steht Ihnen nichts im Wege, sich mit offenen Augen in das Abenteuer Forth zu stürzen. In der folgenden Tabelle 2 finden Sie diesmal nicht die neu besprochenen Befehle, sondern alle diejenigen, die Ihr Forth-System haben sollte, aber nicht haben muß. Die Erklärung der Befehlsworte spornt Sie vielleicht an, die Anweisungen auszuprobieren und in eigenen Programmen zu verwenden. Die Listings in diesem Heft zeigen Ihnen weiter, wie man Forth-Programme entwickelt und realisiert.

(Peter Monadjemi/hg)



=	f = 1 wenn n2 = n1
<	f = 1 wenn n2 < n1
>	f = 1 wenn n2 > n1

Die folgenden Operatoren erwarten eine Zahl n im Top of Stack, welche mit Null verglichen wird.

0 =	f = 1 wenn n = 0
0 <	f = 1 wenn n < 0
0 >	f = 1 wenn n > 0

Tabelle 1. Die Vergleichsoperatoren brauchen die zwei obersten Stellen im Stack

Bild 1. So verändert sich der Stack beim Vergleich

Wort	Beschreibung	Stack-Relation
!	Speichert eine Zahl in der Adresse an oberster Stack-Position	n a -
#	Dient bei der Zahlenausgabe mit Maske für die Zifferndarstellung vorzeichenloser doppelt genauer Integers	$d_1 - d_2$
#>	Beendet die maskierte Zahlenausgabe	d - a n
#IN	Fordert zur Eingabe einer einfach genauen Integer auf	- n
#S	Wandelt bei der Zahlenausgabe mit Maske Ziffernzeichen in den ASCII-Code um	d -
$!	Dient zum Speichern von Strings	$a_\$$ a -
$"	Vereinbart einen String im Arbeitsspeicher	- a
$-TB	Entfernt nachlaufende Blanks vom String	
$.	Druckt einen String	a -
$ARRAY	Vereinbart einen String-Array	$n_1\ n_2 -$
$COMPARE	Vergleicht String-Variable	$a_1\ a_2 - n$
$CONSTANT	Vereinbart eine String-Konstante	
$VARIABLE	Vereinbart eine String-Variable	
$XCG	Vertauscht die Werte in String-Variablen	$a_1\ a_2$ -
'	Liefert die Adresse des nächsten Wortes im Eingabestrom	- a
(	Leitet einen Kommentar ein	
*	Liefert das Produkt zweier Zahlen	$n_1\ n_2 - n$
*/	Multipliziert n_1 mit n_2 und dividiert das doppelt genaue Produkt durch n_3	$n_1\ n_2\ n_3 - n$
*/MOD	Ähnlich wie */; liefert jedoch auch den Rest	$n_1\ n_2\ n_3 - n_r\ n_q$
+	Liefert die Summe zweier Zahlen	$n_1\ n_2 - n$
+!	Inkrementiert den gespeicherten Wert	n a -
+LOOP	Inkrementiert eine Schleifenvariable	n -
,	Compiliert n ins Wörterbuch	n -
-	Subtrahiert n_2 von n_1	$n_1\ n_2 - n$
-TRAILING	Aktualisiert den Zeichenzähler	a n_1
.	Gibt eine Zahl aus	n -
."	Gibt Text aus	
.R	Gibt die Zahl n_1 im Datenfeld n_2 aus	$n_1\ n_2 -$
/	Dividiert n_1 durch n_2	$n_1\ n_2 - n$
/MOD	Division mit Quotient und Rest	$n_1\ n_2 -$
0<	»Wahr« falls n < 0	n - f
0=	»Wahr« falls n = 0	n - f
0>	»Wahr« falls n > 0	n - f
1+	Inkrementiert den obersten Stack-Eintrag um Eins	$n - n_1$
1-	Dekrementiert den obersten Stack-Eintrag um Eins	$n - n_1$
16*	Multipliziert den obersten Stack-Eintrag mit 16	$n - n_1$
2!	Speichert eine doppelt genaue Integer	d a -
2$ARRAY	Definiert einen zweidimensionalen String-Array	$n_1\ n_2\ n_3 -$
2*	Multipliziert die oberste Integer mit 2	$n - n_1$
2+	Addiert 2 auf die oberste Integer	$n - n_1$
2-	Subtrahiert 2 von der obersten Integer	$n - n_1$
2/	Dividiert die oberste Integer durch 2	$n - n_1$
2@	Holt eine doppelt genaue Integer aus der Adresse	a - d
2ARRAY	Definiert einen zweidimensionalen Array	$n_1\ n_2 -$
2CONSTANT	Definiert eine doppelt genaue Konstante	d -
2DARRAY	Definiert einen doppelt genauen Integer-Array	$n_1\ n_2 -$
2DROP	Entfernt die oberste doppelt genaue Integer vom Stack	d -
2DUP	Dupliziert die oberste doppelt genaue Integer auf dem Stack	d - d d
OVER	Dupliziert die zweite doppelt genaue Integer auf dem Stack an oberste Stack-Position	$d_1\ d_2 - d_1\ d_2 d_1$
2ROT	Rotiert die dritte doppelt genaue Integer an oberste Stack-Position	$d_1\ d_2\ d_3 - d_2\ d_3\ d_1$
2SWAP	Vertauscht die obersten beiden doppelt genauen Integers	$d_1\ d_2 - d_2\ d_1$
2VARIABLE	Vereinbart eine doppelt genaue Variable	
:	Leitet die Definition eines FORTH-Wortes ein	
;	Beendet die Definition eines FORTH-Wortes	
<	Wird »wahr« falls $n_1 < n_2$	$n_1\ n_2 - f$
<#	Leitet die Zahleneingabe mit Maske ein	
<=	Wird »wahr«, falls n_1 kleiner oder gleich n_2 ist	$n_1\ n_2 - f$
<>	Wird »wahr«, falls n_1 ungleich n_2 ist	$n_1\ n_2 - f$
<CMOVE	Dupliziert n Speicherwörter beginnend bei a_1 an der Adresse a_2; Übertragung beginnt bei der höchstwertigen Adresse	$a_1\ a_2\ n -$
=	Ist »wahr«, falls n_1 gleich n_2 ist	$n_1\ n_2 - f$
>=	Ist »wahr«, falls n_1 größer oder gleich n_2 ist	$n_1\ n_2 - f$
<IN	Enthält die Startposition für die Untersuchung des Eingabestroms	- a
>R	Überträgt eine Integer auf den Kontroll-Stack; benötigt entsprechendes R>	n -
?DUP	Dupliziert die oberste einfach genaue Integer, es sei denn, diese ist gleich 0	n - n n
@	Holt die an der Adresse gespeicherte einfach genaue Integer	a - n
ABS	Ersetzt die oberste einfach genaue Integer durch ihren Absolutbetrag	$n_1 - n_2$
ALLOT	Erweitert den Speicherbereich einer Variablen um n Byte	n -
AND	Bitweises logisches AND	$n_1\ n_2 - n_3$
ARRAY	Vereinbart einen Array	
ASC	Legt den ASCII-Wert des ersten Zeichens in dem String, der bei a beginnt, auf den Stack	a -
BASE	Enthält die Ein-/Ausgaberadix	a - n
BEGIN	Leitet eine Schleife ein	- a
BLANK	Füllt Speicherbereiche mit Leerzeichen	a n -
BLK	Enthält die Adresse des Blockpuffers für den Eingabestrom	- a
BLOCK	Überträgt den Block n von der Diskette in den Arbeitsspeicher und legt dessen Startadresse auf den Stack	n - a

Wort	Beschreibung	Stack-Relation
BUFFER	Wie BLOCK, die Daten werden jedoch nicht übertragen	n – a
C!	Speichert das niedrigwertige Byte einer einfach genauen Integer	n a –
C@	Holt ein Byte und speichert es als einfach genaue Integer	a – n
CASEND	Beendet eine CASE-Anweisung	
CHR$	Wandelt eine ein Byte lange Integer in ihre ASCII-Darstellung um; das Ergebnis steht im temporären Arbeitsbereich, dessen Adresse auf den Stack gelegt wird	c – a
CMOVE	Überträgt n Bytes von Adresse 1 nach Adresse 2; die Übertragung beginnt bei den niedrigwertigen Adressen	a_1 a_2 n –
COMPILE	Nimmt einen Wert in die Wortdefinition mit auf	
CONSTANT	Vereinbart eine Konstante mit dem Wert n	n –
CONTEXT	Enthält die Adresse des Kontext-Vokabulars	– a
COUNT	Legt die Anfangsadresse des Strings und den String-Zähler auf den Stack	a – a_1 n
CR	Sendet einen Zeilenvorschub	
CREATE	Richtet einen Wörterbucheintrag ein	
CRT	Lenkt die Ausgabe auf den Bildschirm	
CURRENT	Enthält die Adresse des aktuellen Wörterbuches	– a
D#IN	Fordert zur Eingabe einer doppelt genauen Integer auf	– d
D*	Multipliziert doppelt genaue Integers	d_1 d_2 – d_p
D*/	Multipliziert d_1 mit d_2 und dividiert das vierfach genaue Produkt anschließend durch d_3	d_1 d_2 d_3 – d
D*/MOD	Wie D*/; liefert aber auch den Rest	d_1 d_2 d_3 d_r d_q
D+	Addiert zwei doppelt genaue Zahlen	d_1 d_2 – d
D–	Subtrahiert zwei doppelt genaue Zahlen (d_1 minus d_2)	d_1 d_2 – d
D/	Liefert den Quotienten von d_1 und d_2	d_1 d_2 – d
D/MOD	Wie D/, liefert aber auch noch den Rest	d_1 d_2 – d_r d_q
D.	Gibt eine doppelt genaue Integer aus	d –
D.R	Gibt eine doppelt genaue Integer in einem n Zeichen langen Datenfeld aus	d n –
E	Bearbeitet den Block, der vom Inhalt von SCR bestimmt wird	
EDIT	Bearbeitet Block n; n wird in SCR gespeichert	n –
ELSE	Für Programmverzweigungen	c –
EMIT	Gibt ein Zeichen aus	
EMPTY-BUFFERS	Markiert alle Puffer als leer	
ERASE	Setzt n aufeinanderfolgende Byte auf den Wert 0, beginnend mit der Adresse a	a n –
EXECUTE	Führt den Wörterbucheintrag aus, dessen Adresse auf dem Stack liegt	a –
EXIT	Beendet die Programmbearbeitung	

Wort	Beschreibung	Stack-Relation
EXPECT	Liest Zeichen in den Arbeitsspeicher ein, beginnend bei Adresse a, wobei maximal n Zeichen oder bis zum ersten Return gelesen wird	a n –
FILL	Belegt n aufeinanderfolgende Speicherwörter (beginnend bei Adresse a) mit dem ASCII-Wert n_c	a n n_c –
FIND	Sucht die Adresse des nächsten Wortes im Eingabestrom	– a
FLUSH	Speichert die markierten Puffer auf Diskette	
FORGET	Löscht alle Wörter bis einschließlich dem angegebenen aus dem Wörterbuch	–
FORTH	Name des Hauptwörterbuches	
D0=	Ist »wahr«, wenn der doppelt genaue Wert gleich 0 ist	d – f
D<	Ist »wahr«, wenn d_1 kleiner d_2 ist	d_1 d_2 – f
DABS	Liefert den Absolutwert einer doppelt genauen Integer	d_1 – d_2
DARRAY	Vereinbart einen Array mit doppelt genauen Integers	n –
DECIMAL	Setzt die Zahlenbasis auf 10	
DEFINITIONS	Macht den Kontext-Wortschatz zum aktuellen Wortschatz	
DEPTH	Liefert die Stack-Tiefe in Einheiten von einfach genauen Integers	– n
DMAX	Liefert die größere von zwei doppelt genauen Integers	d_1 d_2 – d
DMIN	Liefert die kleinere von zwei doppelt genauen Integers	d_1 d_2 – d
DNEGATE	Dreht das Vorzeichen einer doppelt genauen Integer um	d – –d
DO	Leitet eine Schleife ein	n_1 n_2 –
DRDSECS	Liest Diskettensektoren	a n_1 n_2 n_3 n_4 – nf
DROP	Entfernt die oberste einfach genaue Integer vom Stack	n –
DUP	Dupliziert die oberste einfach genaue Integer	n – n n
DWTSECS	Schreibt Diskettensektoren	a n_1 n_2 n_3 n_4 – nf
HERE	Liefert die Adresse des nächsten verfügbaren Wörterbuch-Bytes	– a
HEX	Umwandlung der Zahlenausgabe in Hexadezimaldarstellung	
HOLD	Zur Einfügung von Zeichen bei der Zahlenausgabe mit Maske	c –
I	Legt den Schleifenindex auf den Stack	– n
I'	Legt den Testwert der Schleife auf den Stack	– n
IF	Für Programmverzweigungen	f –
IMMEDIATE	Schaltet von Compilierung in Ausführung um	n_1 n_2 –
INDEX	Gibt die erste Zeile von n_2 Blocks aus, beginnend mit Block n_1	n_1 n_2 –
J	Liefert den Index der dynamisch übernächsten Schleife auf den Stack	– n
KEY	Legt den ASCII-Code des nächsten Eingabezeichens auf den Stack	– n
L	Gibt den Block aus, dessen Nummer in SCR gespeichert ist	
LEAVE	Beendet eine Schleife	
LEFT$	Überträgt die n ersten Zeichen des Strings, der bei a beginnt, in den temporären Arbeitsbereich	a n – a_1

Tabelle 2. Der Befehlssatz, den Ihr Forth-System haben sollte

Wort	Beschreibung	Stack-Relation
LEN	Legt die Länge eines Strings auf den Stack	a – n
LIST	Gibt den Block n aus und legt n in SCR ab	n –
LITERAL	Nimmt den Stack-Wert, ohne ihn zu interpretieren, in die Compilation mit auf	
LOAD	Lädt den Block n	n –
LOADS	Lädt n_2 Block, beginnend mit n_1	$n_1\ n_2$ –
LOOP	Inkrementiert den Schleifenindex	
M*	Doppelt genaues Produkt zweier einfach genauer Integers	$n_1\ n_2 - d$
M*/	Multipliziert d_1 mit n_2 und speichert das Produkt als dreifach genaue Integer, welche dann durch n_3 dividiert wird; der Quotient ist doppelt genau	$d_1\ n_2\ n_3 - d_2$
M+	Gemischte Addition	$d_1\ n - d_2$
M–	Gemischte Subtraktion	$d_1\ n - d_2$
M/	Gemischte Division	$d\ n_1 - n_2$
M/MOD	Wie M/, außer daß sowohl Quotient als auch Rest geliefert werden	$d\ n_1 - d_r\ n_q$
MAX	Liefert den größeren von zwei Werten	$n_1\ n_2 - n$
MID$	Überträgt an die Adresse a_1 einen n_2 Zeichen langen Teil-String, der ab der $n_1$ten Zeichenposition des Strings a beginnt	$a\ n_1\ n_2 - a_1$
MIN	Liefert den kleineren von zwei Werten	$n_1\ n_2 - n$
MOD	Liefert den Rest der Division von n_1/n_2	$n_1\ n_2 - n$
MOVE	Verschiebt n 16 Byte lange Speicherwörter, beginnend bei a_1, nach a_2	$a_1\ a_2\ n$ –
MYSELF	Erlaubt rekursive Aufrufe	
NCASE	Leitet eine CASE-Anweisung ein	n –
NEGATE	Ersetzt eine Zahl durch die negative Zahl mit dem gleichen Betrag	n – –n
NOT	Negiert ein Flag	$f_1 - f_2$
OCTAL	Setzt die Ein-/Ausgabebasis für Zahlen auf das Oktalsystem	
OTHERWISE	Allgemeiner Ausgang in CASE-Anweisungen	
OVER	Dupliziert die zweite Zahl an oberste Stack-Position	$n_1\ n_2 - n_1\ n_2\ n_1$
PAD	Enthält die Anfangsadresse des temporären Arbeitsbereichs	– a
PAGE	Löscht den Bildschirm	
PCRT	Legt die Ausgabe sowohl auf Bildschirm als auch auf Drucker	
PRINT	Legt die Ausgabe nur auf den Drucker	
QUERY	Für Zeicheneingabe	
QUIT	Löscht den Return-Stack	
R>	Überträgt die oberste einfach genaue Integer vom Return-Stack auf den Parameter-Stack	– n
R@	Dupliziert die oberste einfach genaue Integer vom Return-Stack auf den Parameter-Stack	– n
RANDOMIZE	Initialisiert den Zufallsgenerator	
REPEAT	Für die Programmierung von Schleifen	
RIGHT$	Überträgt die n letzten Zeichen des Strings a in den temporären Arbeitsbereich, liefert dessen Adresse	$a\ n - a_1$
RN1	Erzeugt eine Zufallszahl und speichert sie in SEED	
RND	Erzeugt eine Zufallszahl zwischen 1 und n_1	$n_1 - n_2$
ROLL	Legt die n-te einfach genaue Integer auf dem Stack an oberste Stack-Position	$n_0 - n_n$
ROT	Befördert die dritte einfach genaue Integer an oberste Stack-Position	$n_1\ n_2\ n_3 - n_2\ n_3\ n_1$
SAVE-BUFFERS	Markiert alle Puffer für nachfolgende Sicherungen	
SCR	Enthält die Adresse des zuletzt bearbeiteten Blockpuffers	– a
SIGN	Fügt den ASCII-Code des Minuszeichens bei Zahlenausgabe mit Maske ein, falls n negativ ist	n –
SPACE	Gibt ein Leerzeichen aus	
SWAP	Vertauscht die beiden obersten Stack-Einträge	$n_1\ n_2 - n_2\ n_1$
THEN	Bei Programmverzweigungen benötigt	
TYPE	Gibt n Zeichen beginnend ab der Adresse a aus	a n –
U*	Vorzeichenlose Integermultiplikation	$u_1\ u_2 - u$
U.	Gibt eine vorzeichenlose Integer aus	u –
U.R	Gibt eine vorzeichenlose Integer in einem n Stellen breiten Datenfeld aus	u n –
U/MOD	Vorzeichenlose Division mit doppelt genauem Dividenden, liefert Quotienten und Rest	$u_d\ u_1 - u_r\ u_q$
U<	»Wahr« falls u_1 kleiner u_2 ist (vorzeichenlose Integers)	$u_1\ u_2$ –
UNTIL	Für die Programmierung von Schleifen	f –
UPDATE	Markiert alle Blockpuffer als gesichert	
VARIABLE	Definiert eine Variable	
VOCABULARY	Für die Vereinbarung eines neuen Wortschatzes	
WHILE	Für die Programmierung von Schleifen	f –
WORD	Liest Zeichen aus dem Eingabestrom; Trenner ist Zeichen mit ASCII-Code n	n – a
XOR	Bitweises exklusives ODER	$n_1\ n_2 - n$
Y/N	Fragt nach Y oder N; N liefert Wahrheitswert »wahr«	– f
[	Beendet Compilierung und leitet Ausführung ein; wird in Wortdefinition benötigt	
[COMPILE]	Bewirkt, daß ein Wort mit dem Status IMMEDIATE compiliert wird	
]	Beendet Ausführung und fährt mit der Compilierung fort	

Tabelle 2. Der Befehlssatz, den Ihr Forth-System haben sollte (Schluß)

Tabelle aus »Der Einstieg in Forth«, Markt&Technik Verlag AG, ISBN 3-89090-085-2

Die Buchstaben der Stack-Relationsspalte bedeuten:

a = Adresse
c = ASCII-Code
d = doppeltgenaue Zahl
f = Flag
n = ganze Zahl
r = Rest (bei Division)
q = ganzzahliges Ergebnis (bei Division)

Professionelle Grafikprogramme für Schneider CPC 6128 + Joyce

DIGITAL RESEARCH®

DR Draw

DR Draw: Macht aus Ihren Ideen ein Kunstwerk

Verwenden Sie DR Draw, um Organisations-Diagramme, Flußdiagramme, Logos, technische Zeichnungen, Schaubilder, Platinenentwürfe und jede nur erdenkliche Art von Linien- und Formgrafiken zu entwerfen. Und jeder Bestandteil Ihrer Zeichnung kann auf vielfältige Weise durch Farben und Schraffuren hervorgehoben werden.

Einfachste Bedienung

DR Draw verwendet leichtverständliche Menüs zur Steuerung seiner Funktionen und Erstellung einer Zeichnung. Sie können aus vorprogrammierten Figuren wie Kreisen, Quadern, Rechtecken, Kreisbögen, Polygonen und Linien auswählen oder Ihre eigenen Figuren entwerfen oder die bestehenden verändern. An beliebigen Stellen kann erläuternder Text in eine Zeichnung eingefügt werden. Außerdem haben Sie die Wahl zwischen mehreren Schriftarten.

Flexibilität bei der Gestaltung

Jeder Teil einer Zeichnung kann auf Tastendruck überarbeitet und verändert werden: Figuren können mit Farben oder Mustern gefüllt werden; sie können vergrößert oder verkleinert oder an eine neue Position verschoben oder kopiert werden. Ebenso können die Schriftarten, Größen, Farben und Positionen mit wenigen Tastendrücken geändert werden.

Vergrößerungen und Ausschnittdarstellungen

Mit Hilfe einer besonderen Funktion von DR Draw können Sie Einzelheiten Ihrer Zeichnung vergrößern, um Details besser bearbeiten zu können.

Ausgabe auf Papier, Transparentfolie oder Film

Was immer Sie erstellen, kann gespeichert oder zu Berichts- und Präsentationszwecken auf Papier, Transparentfolie oder Film geplottet oder gedruckt werden. DR Draw druckt Ihre Zeichnung exakt auf eine DIN-A4-Seite.

Hardwarevoraussetzungen

DR Draw läuft auf jedem Schneider CPC 6128 oder Joyce PCW 8256 mit einem oder zwei Diskettenlaufwerken. Die Grafiken können auf jedem Drucker oder Plotter ausgegeben werden, für den ein GSX-Treiber verfügbar ist. Dazu zählen Schneider-, Epson- und Shinwa-Drucker sowie der Plotter HP 7470A.

Die Fähigkeiten auf einen Blick

- Erstellung beliebiger Zeichnungen
- vorprogrammierte Figuren wie Kreise, Quader, Rechtecke, Kreisbögen, Polygone und Linien
- freie Wahl der Gestaltungselemente wie Farben, Muster und Schriftarten
- Vergrößerungen und Ausschnittdarstellungen
- Teile einer Zeichnung können kopiert, verschoben oder gelöscht werden
- Grafiken können gespeichert, geplottet oder gedruckt werden
- einfache Bedienung durch Menüauswahl

Best.-Nr. MS 613 **DM 199,–*** (sFr. 178,–/öS 1890,–*)

DR Graph

DR Graph: Präsentationsgrafiken mit professionellem Niveau

DR Graph ist ein interaktives Softwarepaket, mit dem Sie Ihren Mikrocomputer zur Erstellung von Geschäftsgrafiken und Text-Charts verwenden können. DR Graph macht es leicht, komplexe geschäftliche oder wissenschaftliche Daten in übersichtliche und aussagekräftige Grafiken zu verwandeln.

Ein Bild sagt mehr als tausend Worte

Eine gut dargestellte Grafik weckt das Interesse und die Aufmerksamkeit des angezielten Personenkreises eher als andere Kommunikationsarten. Grafisch dargestellte Fakten können leichter analysiert, verstanden und behalten werden.

Einfachste Bedienung

Mit DR Graph können Sie die Grafik dem Computer schnell und leicht beschreiben. Zur Erstellung einer Grafik werden die gewünschten Optionen ganz einfach aus übersichtlichen Menüs ausgewählt. DR Graph kann von jedermann bedient werden, der mit einfachen Grundlagen der Mikrocomputerbedienung vertraut ist.

Flexibilität bei der Gestaltung

Zusätzlich zur vorhandenen Computerschrift stehen drei verschiedene Schriften für Titelzeilen, Legenden und Anmerkungen zur Verfügung. Auch bei der Gestaltung der Grafiken kann aus zahlreichen Linientypen, Linien- und Balkenbreiten und acht Schraffuren gewählt werden.

Ansehen, speichern und drucken

Mit DR Graph können Sie auf dem Bildschirm immer genau sehen, wie Sie Ihre Grafik gestalten. Anschließend können Sie sie drucken oder auf Diskette speichern, um sie später weiter zu bearbeiten.

Hardwarevoraussetzungen

DR Graph läuft auf jedem Schneider CPC 6128 oder Joyce PCW 8256 mit einem oder zwei Diskettenlaufwerken. Die Grafiken können auf jedem Drucker oder Plotter ausgegeben werden, für den ein GSX-Treiber verfügbar ist. Dazu zählen Schneider-, Epson- und Shinwa-Drucker sowie der Plotter HP 7470A.

Die Fähigkeiten auf einen Blick

- Linien-Grafiken, Histogramme, Torten-Grafiken, Stufen-Grafiken, Strich-Histogramme, Punkte-Grafiken und Text-Grafiken
- freie Wahl der Gestaltungselemente wie Beschriftungen, Titelzeilen, Legenden, Farben, Schriftarten und Ränder
- frei wählbare Skalierung
- variable Linien- und Balkenbreite
- Schnittstelle zu anderen Programmen
- beliebig positionierbare Anmerkungen
- Grafiken können gespeichert, geplottet oder gedruckt werden
- einfache Bedienung durch Menüauswahl

Best.-Nr. MS 614 **DM 199,–*** (sFr. 178,–/öS 1890,–*)

In Vorbereitung:

Fakturierung

Ein dBASE-II-Anwenderprogramm, das folgende Möglichkeiten bietet: Angebotsschreibung und Rechnungsschreibung, Artikelverwaltung, Adreßverwaltung, Nachkalkulation. Der dokumentierte Quellcode wird für individuelle Programmanpassungen mitgeliefert.

Best.-Nr. MS 616 **DM 94,–*** (sFr. 82,–/öS 940,–*)

Finanz-Buchhaltung

Das Komplett-Paket für den Schneider CPC 6128 und Joyce. Erstellen von Kontenplänen, Umsatzsteuerauswertung und Einnahmen-/Überschußrechnung. Betriebswirtschaftliche Auswertungen wie Journalschreibung und Kostenstellenrechnung möglich.

Best.-Nr. MS 615 **DM 194,–*** (sFr. 175,–)

* Inkl. MwSt. Unverbindliche Preisempfehlung



Unternehmensbereich Buchverlag

Hans-Pinsel-Straße 2, 8013 Haar bei München

Diese Markt&Technik-Softwareprodukte erhalten Sie in den Fachabteilungen der Kaufhäuser und in Computershops.

Bestellungen im Ausland bitte an untenstehende Adressen.

Schweiz: Markt&Technik Vertriebs AG, Kollerstr. 3, CH-6300 Zug, Tel. 042/41 56 56

Österreich: Ueberreuter Media Handels- und Verlagsges. mbH, Alser Straße 24, A-1091 Wien, 0222/48 15 38-0

Forth zum Abtippen

Gelerntes will auch geübt werden. Wer noch keinen Forth-Interpreter hat, der findet hier einen, der nichts kostet. Einfach eintippen, RUN eingeben und mit der ENTER-Taste starten.

Nachdem Sie sich jetzt durch viele Seiten Forth hindurchgekämpft haben, wollen Sie Ihre Forth-Programme zum Laufen bringen. Doch gleich einen Compiler kaufen, das muß nicht sein. Testen Sie erst einmal Ihr Interesse mit diesem kostenlosen Basic-Forth-Interpreter.

»Basic-Forth V.4« ist vollständig in Basic geschrieben und läuft auf beinahe jedem Computer. Spezielle Befehle wurden fast völlig weggelassen und erklären sich, wenn, durch ihre Anweisungen. Also einfach eingetippt und schon beginnt Ihr Forth-Vergnügen.

Die Profis unter Ihnen werden jetzt sicher schmunzeln. Forth-Interpreter in Basic – da geht doch der ganze Geschwindigkeitsgewinn in die Binsen. Richtig, aber mit Basic-Forth sollen Sie nicht professionell programmieren, sondern ausprobieren. Und da ist Basic zum Eingeben eben die leichteste Programmiersprache. Wenn dann erst einmal Interesse an dieser Sprache geweckt ist, können Sie immer noch auf einen echten Forth-Compiler umsteigen. (hg)

```
1 CLS
2 DIM S(80),R(80),L(80),LO(80),I$(800)
3 DIM B$(80)
4 PRINT " BASIC-FORTH     V.4"
20 REM
24 ON ERROR GOTO 29
28 GOTO 30
29 PRINT A$,"?"
30 M=0
32 N=0
60 K=1
62 INPUT I$
63 I$=I$+" "
64 L1=0
70 L(K)=L1
72 LO(K)=LEN(I$)
74 L1=LO(K)
100 IF N<0 THEN GOTO 106
104 GOTO 110
106 PRINT "STACK EMPTY"
108 GOTO 30
110 L(K)=L(K)+1
112 IF L(K)>LO(K) THEN GOTO 132
114 B$=MID$(I$,L(K),1)
116 IF B$=" " THEN GOTO 110
118 A$=B$
120 L(K)=L(K)+1
122 B$=MID$(I$,L(K),1)
124 IF B$=" " THEN GOTO 130
126 A$=A$+B$
128 GOTO 120
130 GOTO 200
132 IF K<2 THEN GOTO 60
134 K=K-1
135 I$=MID$(I$.1,LO(K))
136 L1=LO(K)
138 GOTO 110
200 REM    DICTIONARY
300 IF A$<>"SQUARE" THEN GOTO 310
302 B$="DUP * "
304 I$=I$+B$
306 K=K+1
308 GOTO 70
310 IF A$<>"CUBE" THEN GOTO 320
312 B$="DUP SQUARE * "
314 I$=I$+B$
316 K=K+1
318 GOTO 70
320 IF A$<>"TEST" THEN GOTO 330
322 B$="DO PI 10 / R@ * SIN . LOOP "
324 I$=I$+B$
326 K=K+1
328 GOTO 70
330 REM
902 IF A$<>"+" THEN GOTO 910
904 N=N-1
906 S(N)=S(N)+S(N+1)
908 GOTO 100
910 IF A$<>"-" THEN GOTO 920
912 N=N-1
914 S(N)=S(N)-S(N+1)
916 GOTO 100
920 IF A$<>"*" THEN GOTO 930
922 N=N-1
924 S(N)=S(N)*S(N+1)
926 GOTO 100
930 IF A$<>"/" THEN GOTO 940
932 N=N-1
934 S(N)=S(N)/S(N+1)
936 GOTO 100
940 IF A$<>"ABS" THEN GOTO 950
942 S(N)=ABS(S(N))
944 GOTO 100
950 IF A$<>"ATN" THEN GOTO 960
952 S(N)=ATN(S(N))
954 GOTO 100
960 IF A$<>"COS" THEN GOTO 970
962 S(N)=COS(S(N))
964 GOTO 100
970 IF A$<>"EXP" THEN GOTO 980
972 S(N)=EXP(S(N))
974 GOTO 100
980 IF A$<>"INT" THEN GOTO 990
982 S(N)=INT(S(N))
984 GOTO 100
990 IF A$<>"LOG" THEN GOTO 1000
992 S(N)=LOG(S(N))
994 GOTO 100
1000 IF A$<>"RND" THEN GOTO 1010
1002 S(N)=RND(-N)
1004 GOTO 100
1010 IF A$<>"SGN" THEN GOTO 1020
1012 S(N)=SGN(S(N))
1014 GOTO 100
1020 IF A$<>"SIN" THEN GOTO 1030
1022 S(N)=SIN(S(N))
1024 GOTO 100
1030 IF A$<>"SQR" THEN GOTO 1040
1032 S(N)=SQR(S(N))
```

Listing. Ein Forth-Interpreter zum Abtippen

```
1034 GOTO 100
1040 IF A$<>"TAN" THEN GOTO 1050
1042 S(N)=TAN(S(N))
1044 GOTO 100
1050 IF A$<>"-" THEN GOTO 1060
1052 S(N)=S(N)-S(N+1)
1054 GOTO 100
1060 IF A$<>"S?" THEN GOTO 1070
1062 FOR I=1 TO N
1064 PRINT S(N-I+1)
1066 NEXT I
1068 GOTO 100
1070 IF A$<>"." THEN GOTO 1080
1071 IF N<1 THEN GOTO 106
1072 PRINT S(N)
1074 N=N-1
1076 GOTO 100
1080 IF A$<>"DUP" THEN GOTO 1090
1082 N=N+1
1084 S(N)=S(N-1)
1086 GOTO 100
1090 IF A$<>"DROP" THEN GOTO 1100
1092 N=N-1
1094 GOTO 100
1100 IF A$<>"SWAP" THEN GOTO 1110
1102 S(N+1)=S(N-1)
1104 S(N-1)=S(N)
1106 S(N)=S(N+1)
1108 GOTO 100
1110 IF A$<>"OVER" THEN GOTO 1120
1112 N=N+1
1114 S(N)=S(N-2)
1116 GOTO 100
1120 IF A$<>">R" THEN GOTO 1130
1122 M=M+1
1124 R(M)=S(N)
1126 N=N-1
1128 GOTO 100
1130 IF A$<>"R>" THEN GOTO 1140
1132 N=N+1
1134 S(N)=R(M)
1136 M=M-1
1138 GOTO 100
1140 IF A$<>"R@" THEN GOTO 1200
1142 N=N+1
1144 S(N)=R(M)
1146 GOTO 100
1200 REM
1202 IF A$<>"=" THEN GOTO 1210
1203 N=N-1
1204 IF S(N)=S(N+1) THEN GOTO 1207
1205 S(N)=0
1206 GOTO 100
1207 S(N)=1
1209 GOTO 100
1210 IF A$<>">" THEN GOTO 1220
1212 N=N-1
1214 IF S(N)>S(N+1) THEN GOTO 1217
1215 S(N)=0
1216 GOTO 100
1217 S(N)=1
1218 GOTO 100
1220 IF A$<>"<" THEN GOTO 1230
1222 N=N-1
1223 IF S(N)<S(N+1) THEN GOTO 1227
1224 S(N)=0
1225 GOTO 100
1227 S(N)=1
1228 GOTO 100
1230 IF A$<>"IF" THEN GOTO 1250
1231 N=N-1
1232 IF S(N+1) THEN GOTO 100
1233 FOR I=L(K) TO LO(K)-3
1234 B$=I$(I,I+3)
1235 IF B$="ELSE" THEN GOTO 1240
1236 IF B$="THEN" THEN GOTO 1240
1237 NEXT I
1238 PRINT "IF?"
1239 GOTO 30
1240 L(K)=I+4
1241 GOTO 100
1242 GOTO 100
1250 IF A$<>"ELSE" THEN GOTO 1260
1252 GOTO 1233
1260 IF A$<>"THEN" THEN GOTO 1270
1262 GOTO 100
1270 IF A$<>"BEGIN" THEN GOTO 1280
1272 M=M+1
1274 R(M)=L(K)
1276 GOTO 100
1280 IF A$<>"UNTIL" THEN GOTO 1300
1282 N=N-1
1283 IF S(N+1) THEN GOTO 1288
1284 IF S(N+1) THEN GOTO 100
1286 L(K)=R(M)
1287 GOTO 100
1288 M=M-1
1289 GOTO 100
1300 IF A$<>"DO" THEN GOTO 1320
1302 M=M+1
1304 R(M)=L(K)
1305 M=M+1
1306 R(M)=S(N-1)
1308 M=M+1
1309 R(M)=S(N)
1310 N=N-2
1312 GOTO 100
1320 IF A$<>"LOOP" THEN GOTO 1340
1322 R(M)=R(M)+1
1324 IF R(M-1)>R(M) THEN GOTO 1330
1326 M=M-3
1328 GOTO 100
1330 L(K)=R(M-2)
1332 GOTO 100
1340 REM
1500 IF A$<>"PI" THEN GOTO 1510
1502 N=N+1
1504 S(N)=3.14159
1506 GOTO 100
1510 IF A$<>"0" THEN GOTO 1520
1512 N=N+1
1514 S(N)=0
1516 GOTO 100
1520 IF A$<>"STOP" THEN GOTO 1600
1522 STOP
1600 NUM=1
1602 FOR I=1 TO LEN(A$)
1604 IF MID$(A$,I,1)<"0" OR MID$(A$,I,1)
>"9" THEN NUM=0
1606 IF I=1 AND MID$(A$,1,1)="-" THEN NU
M=1
1608 NEXT I: IF NUM=0 THEN PRINT A$;"  N
OT DEFINED": GOTO 30
1610 N=N+1
1612 S(N)=VAL(A$)
1614 GOTO 100
```

Listing. Ein Forth-Interpreter zum Abtippen (Schluß)

Trace-Befehl für FIG-Forth

Hier wird ein neues, nützliches Forth-Wort vorgestellt, mit dem man den Ablauf anderer Wörter schrittweise untersuchen kann.

Die einfachste Art, ein Forth-Wort zu testen, besteht darin, es mit den entsprechenden Eingangswerten auf dem Stack aufzurufen und dann zu hoffen, daß sich kein Fehler eingeschlichen hat. Wenn sich der Computer dann noch normal zurückmeldet, der richtige Wert ausgegeben wird und der Stack sich in dem Zustand befindet, in dem er sich auch befinden sollte, so kann angenommen werden, daß das Wort richtig arbeitet. Manchmal aber — und das kommt öfter vor, als man möchte — verliert sich der Computer in einem Irrgarten und nichts läuft mehr. In diesem Falle ist es sehr nützlich, ein Wort zur Verfügung zu haben, das etwa einem TRACE-Befehl in Basic entspricht. Solch ein TRACE ist aber nicht ganz einfach zu realisieren. Der Grund dafür ist unter anderem darin zu suchen, daß es in Forth nicht nur Wörter gibt, die nur 2 Byte Platz in dem Parameterfeld beanspruchen, sondern auch ein paar andere, die neben ihrer CFA (Code-Feld-Adresse) auch noch Daten ablegen. Dazu gehören alle strukturierenden Wörter (IF, ELSE, THEN / DO, LOOP, +LOOP / BEGIN, UNTIL, WHILE, REPEAT, AGAIN), die entweder BRANCH oder auch 0BRANCH compilieren und daran noch ihre Sprungweite anhängen. Ebenso zählen LIT und CLIT dazu, die außer ihrer CFA noch den Wert der Konstanten eintragen. Nicht zu vergessen auch das Wort »."«, das ganze Texte in das Parameterfeld speichert.

Alle Wörter, die in irgendeiner Weise den Returnstack manipulieren, sind mit besonderer Vorsicht zu behandeln, da ein TRACE-Wort diesen Stack selber benötigt. Die Wörter R〉, R, DO, LOOP, +LOOP, I, I', J, K, LEAVE gehören dazu und müssen deshalb von TRACE alle gesondert behandelt werden. Man muß für jedes dieser Wörter eine Routine schreiben, die an einem eigenen Stack diese Manipulationen entsprechend dem originalen Programm durchführt. Alle angegebenen Wörter müssen von »TRACE« gesondert behandelt werden. Der ganze Rest aber kann von dem internen Interpreter ausgeführt werden. Für sämtliche Returnstack-Manipulationen benötigt man neben einem eigenen Returnstack auch noch den entsprechenden Pointer.

ONE-STEP ist kein neuer Tanzschritt

Am einfachsten erscheint es deshalb, TRACE als Rahmenprogramm aufzufassen, das die Ein- und Ausgaben durchführt und mit dem Anwender kommuniziert. Muß dann einmal ein Wort ausgeführt werden, wird ONE-STEP (Listing) aufgerufen, das dann das Wort ausführt, auf das der selbst definierte Instruction-Pointer zeigt.

In dem Wort ONE-STEP sind dann sämtliche Fälle, die nicht von dem inneren Interpreter ausgeführt werden können, einzeln abzuarbeiten.

Mit dem Wort ONE-STEP können fast alle Forth-Wörter getestet werden. Falls es sich bei dem Wort um ein Primitive, eine Konstante oder Variable handeln sollte, so wird Ihnen das sofort mitgeteilt. Nur Forth-Wörter, die in Highlevel verfaßt sind, werden von TRACE auch entsprechend behandelt. Wenden Sie TRACE auch mal auf Wörter des Kernals an. Dadurch erhalten Sie einen guten Einblick in die Arbeitsweise von TRACE, und Sie lernen so auch sehr gut die Programmierung in Forth selbst kennen.

Hier ein paar Beispiele:

```
4 TRACE ·
: · S-〉D D.4 ;
4 . TRACE D.
: D. 0 D.R SPACE 4 ;
4 . 0 TRACE D.R
: D.R 〉R SWAP OVER DABS 〈#   #S SIGN #〉 R〉 OVER
SPACES TYPE 4 ;
```

Sie sehen daran, wie eng verknüpft Forth selbst in so einem grundlegenen Wort wie ».« ist. Auch die Decompilereigenschaften von Forth sind hier ein wenig dargestellt.

TRACE funktioniert so lange als Decompiler, bis ein Wort auszuführen ist, das in seinem Verlauf eine Ausgabe durchführt. Diese Aufgabe, die normalerweise als einzige auf dem Bildschirm erscheinen würde, steckt jetzt mitten in dem entschlüsselten Wort und macht es so manchmal etwas problematisch, den genauen Sourcetext zu erkennen. Weiterhin werden alle strukturierenden Wörter nicht angezeigt, sondern nur deren »Run Time Executive« (BRANCH oder 0BRANCH) und eventuell auch ausgeführt.

Es kann sein, daß die Version Ihres Forth nicht ganz genau mit der übereinstimmt, die wir eingesetzt haben. So ist zum Beispiel das Wort CLIT nicht in jeder Version implementiert. Es kann sein, daß Sie das Wort DLIT in Ihrem Programm vertreten haben.

Einfache Anpassung

Für den Fall, daß Sie kein CLIT haben, lassen Sie die ganze Zeile einfach unter den Tisch fallen und tippen gleich ein THEN weniger ein.

Ist das Wort DLIT aber in Ihrer Version enthalten, so müßten Sie einfach eine neue Zeile einfügen, ganz der Zeile von CLIT entsprechend. Der Unterschied zu CLIT besteht in der Änderung von: 1. Lesebefehl C@ nach D@ (Liest statt einem Byte ein Langwort, 32 Bit), 2. IPOI darf nicht nur um 1 erhöht, sondern muß um 4 inkrementiert werden.

In dem Falle, daß sonst noch irgendwelche Wörter in Ihrem Forth vorhanden sind, die Daten mit in das Parameterfeld mit ablegen (dies kommt oft bei Erweiterungen vor, Strings, spezielle Datenwörter), so müßten Sie diese Wörter selber behandeln. Entsprechendes gilt auch für die Returnstack-manipulierenden Wörter.

Nicht anwenden sollten Sie TRACE bei Wörtern, die selbst nur zur Definition von anderen Datentypen entwickelt wurden. Insbesondere die 〈BUILDS .... DOES〉-Funktion funktioniert nicht ganz vollständig mit unserem »Trace«. Auch mit anderen Wörtern mag es Schwierigkeiten geben, doch diese Wörter befinden sich in der Unterzahl.

Noch ein letzter Hinweis. Der Stackpointer muß nach ordnungsgemäßer Beendigung des Wortes auf 0 stehen. Jeder andere Wert in SPOI würde bei normalem Ablauf durch den inneren Interpreter zum Absturz des Systems führen. Denken Sie bitte auch daran: Wenn das Wort durch »;« beendet wird, muß in SPOI eine Null stehen. Deshalb lasse ich den Wert dieser Variablen beim Abbruch auch gleich mit ausdrucken.

Für etwaige Anregungen oder Verbesserungsvorschläge sind wir dankbar.

(Bernhard Leikauf/ev)

```
0 VARIABLE IPOI                           (Def. Instruction Pointer)
0 VARIABLE SPOI                           (Def. Returnstack Pointer)
0 VARIABLE RSTA 38 ALLOT                  (Def. Returnstack für max. 20 Einträge)
: ONE-STEP IPOI @                         (Hole IP aus IPOI)
DUP @ ' OBRANCH CFA =                     (Wort = OBRANCH)
 IF SWAP IF DROP 2 ELSE                   (Nimm Flag vom Stack)
2+ @ THEN IPOI +! ELSE                    (True Flag. Dann überspringe den Offset False Flag. Addiere den Offset zu IPOI;)
                                          (führe den Sprung aus)
DUP @ ' BRANCH CFA =                      (Wort = BRANCH)
 IF  2+ @ IPOI +! ELSE                    (Addiere den Offset zu IPOI; führe den Sprung immer durch)
DUP @ ' LIT CFA =                         (Wort = LIT+)
 IF 2+ @ DUP . 2 IPOI +! ELSE             (Hole den 16-Bit-Wert, zeige ihn an und lege ihn auch auf dem Stack ab. Setze IPOI auf das)
                                          (Wort nach dieser Zahl)
DUP @ 'CLIT CFA =                         (Wort = CLIT)
IF 2+ C@ DUP . 1 IPOI +! ELSE             (Hole den 8-Bit-Wert, zeige ihn an und lege ihn auch auf dem Stack ab. Setze IPOI auf das)
                                          (Wort nach dieser Zahl)
DUP @ ' (.'') CFA =                       (Wort = (.''))
 IF 2+ COUNT DUP 1+ IPOI +! TYPE 32 EMIT  (Versetze IPOI auf das Wort hinter dem Text und drucke diesen Text mit »Space« aus)
ELSE
DUP @ ' )R  CFA =                         (Wort = )R)
 IF DROP RSTA SPOI @+ !                   (Speichere den Wert auf dem Stack an die)
2 SPOI+! ELSE                             (Adresse, auf die der Stackpointer zeigt und erhöhe dann den Stackpointer um 2)
DUP @ ' R  CFA = OVER @ ' I CFA = OR      (Wort = R oder = I  ? dasselbe PRG)
IF DROP RSTA SPOI @+ 2- @ ELSE            (Kopiere den Wert, auf den der Stackpointer zeigt, auf den Parameterstack)
DUP @ ' R)  CFA =                         (Wort = R) )
 IF DROP RSTA SPOI @+ 2- @-2 SPOI +!      (Hole den obersten Eintrag des Returnstacks auf den P.stack und dekrementiere den)
ELSE                                      (Stackpointer SPOI um 2)
DUP @ ' (DO)  CFA = IF DROP SWAP RSTA     (Speichere den Schleifenindex und das)
SPOI @+ 2! 4 SPOI +! ELSE                 (Maximum auf dem eigenen R.stack ab)
DUP @ ' I'  CFA =                         (Wort = I')
 IF DROP RSTA SPOI @ + 4 - @ELSE          (Hole den zweitobersten Eintrag auf den P.stack)
DUP @ 'J  CFA =                           (Wort = J)
 IF DROP RSTA SPOI @ + 6 - @ELSE          (Hole den dritten Eintrag des R.stacks)
DUP @ 'K  CFA =                           (Wort = K)
 IF DROP RSTA SPOI @ + 10 - @ELSE         (Hole den fünften Eintrag des R.stacks)
DUP @ ' LEAVE  CFA =                      (Wort = LEAVE)
 IF DROP RSTA SPOI @ + 2 - DUP @ SWAP 2-  (Ändere das Max. der Schleife auf den gegenwärtigen Wert des Schleifenindex (I) ab)
! ELSE
DUP @ ' (LOOP)  CFA =                     (Wort) = (LOOP)
 IF RSTA SPOI @ + 2- DUP 1+! DUP 2- @     (Inkrementiere den Schleifenindex. Hat er das Max. erreicht?)
SWAP @)
 IF 2+ @ IPOI +!                          (Führe Sprung zum Schleifenstart aus, wenn I' ) I)
ELSE DROP -4 SPOI +! 2 IPOI +!            (Schleifenende erreicht. Verringere Stackpointer)
THEN ELSE                                 (um 4 (Index und Max.. Dann überspringe den Offset zum Schleifenanfang)
DUP @ ' (+LOOP) CFA =                     (Wort = (+LOOP))
 IF RSTA SPOI @ + 2- ROT OVER +!DUP 2- @  (Erhöhe den Schleifenparameter um den angegebenen Wert)
SWAP @)                                   (Ist jetzt das Maximum erreicht oder schon überschritten?)
 IF 2+ @ IPOI +!                          (Selbe Operation wie bei (LOOP))
ELSE DROP -4 SPOI +! 2 IPOI +!
THEN  ELSE                                (Eventuelle weitere Abweichungen von der Norm wären dann hier fortlaufend einzutragen)
@ EXECUTE                                 (Andernfalls ist es ein vom inneren Interpreter)
                                          (ausführbares Wort und so auch von diesem zu erledigen)
THEN THEN THEN THEN THEN THEN THEN THEN
THEN THEN THEN THEN THEN THEN THEN
2 IPOI +!                                 (Abschluß sämtlicher eröffneter Abfragen, stelle nach Erledigen des Wortes IPOI auf)
                                          (das nächste zu erledigende Wort).

: TRACE  -Find
   IF DROP CFA DUP @ ' ONE-STEP CFA @ =
     IF CR .'' : '' 2+ DUP NFA ID. IPOI ! 0 SPOI !
       BEGIN IPOI @ @ DUP 2+ NFA ID.
            BEGIN (Warteschleife des Rechners) UNTIL
            ' ;S CFA = ?TERMINAL OR IF CR SPOI ? [COMPILE],S THEN
            ONE-STEP
       AGAIN
     ELSE .'' NO HIGH LEVEL '' DROP THEN
   ELSE .'' NOT FOUND '' THEN ;
```

Listing. Das vollständige »Trace«-Programm. Die Kommentare sind nicht einzugeben.

Turtle-Grafik mit Forth

Als Beispiel für ein komplexes Programm in Forth stellen wir hier ein Grafik-Paket vor.

Obwohl ursprünglich für den C 64 mit HES-Forth geschrieben, lohnt es sich sicher auch für die Besitzer anderer Computer oder anderer Forth-Versionen, sich mit diesem Grafikpaket etwas näher zu beschäftigen. HES-Forth ist im wesentlichen ein etwas erweitertes FIG-Forth. Zur Anpassung an andere Computer brauchen nur die systemspezifischen Teile dieses Programm-Pakets geändert werden.

Im folgenden Text geben wir eine ausführliche Programmbeschreibung, die die Arbeit mit diesem Programm erleichtern soll. Das gesamte Paket besteht aus drei Teilen:
- dem High-Resolution-Graphic-Package
- der Multi-Color-Graphic
- den Extras

Das Programm wird durch die Lade-Screens 02, 03 und 04 des Main-File geladen.

Es steht eine hochauflösende Grafikseite mit einer Auflösung von 320 x 200 Pixel zu Verfügung. Der Punkt (0,0) liegt dabei in der linken oberen Ecke. Die Befehle HION und HIOFF schalten zwischen dem normalen Arbeitsbildschirm und der Grafikseite hin und her. Alle Eingaben können weiterhin im direkten Modus erfolgen. Ein Beispiel sieht so aus:

```
HION               ( Einschalten der Grafik )
14 0 CFILL         ( Farben einstellen )
HCLEAR             ( Bildschirm löschen )
0 0 319 199 LINK   ( Diagonale ziehen )
HIOFF              ( Ausschalten der Grafik )
```

Entsprechend der üblichen Umgekehrt Polnischen Notation erwarten alle Worte ihre Parameter auf dem Stack. Wichtig sind dabei immer die Leerzeichen (Spaces) zwischen den Wörtern. Solange die Grafikseite aktiv ist, sind die Eingaben ja nicht sichtbar.

Bei einem Tippfehler drückt man entweder RUN/RESTORE – dabei wird der Stack und der Arbeitsbildschirm gelöscht, nicht aber die Grafikseite – und wiederholt die letzte Eingabe. Oder aber man gelangt durch HIOFF zurück auf den Arbeitsbildschirm, der die letzten Eingaben zeigt. Der Handler zur Adreßberechnung für X-Werte außerhalb (0/319) und Y-Werte außerhalb (0/199) steht in XFORM beziehungsweise YFORM und kann vom Benutzer abgeändert werden. XFORM und YFORM sind als »wrap-around« initialisiert, das heißt (-1,-1) ist gleich (319,319) etc.

Die Grafik kann als sequentielle Datei auf Diskette gespeichert werden, und zwar mittels der Befehle HIWRITE und HIREAD. Zuvor ist einmal mit »SEQ name« ein Filename festzulegen. Die beiden Befehle beziehen sich so lange auf name, bis dieser mit SEQ geändert wird.

Der Aufbau der Datei sieht folgendermaßen aus:

8 KByte	Bit-Maß
1 Byte	Hintergrundfarbe
1000 Byte	Farb-RAM (low)
1000 Byte	Farb-RAM (high)

PLOT und LINK stehen sowohl als High-Level-Wort wie auch als Primitive zur Verfügung. Welche Version geladen werden soll, wird mit Scr # 02 eingestellt.

Nun zum Laden:

```
FILE MAIN
2 LOAD
FCLOSE
```

Der Ladevorgang nimmt etwa vier Minuten in Anspruch.

Zusätzlich zur normalen hochauflösenden Grafik bietet das Paket eine Multi-Color-Grafik mit 160 x 200 doppelt-breiten Punkten, wobei jedes Pixel eine von vier möglichen Farben haben kann. Das Ein-/Ausschalten nehmen MON (Multi-Color ein) und MOFF (aus) vor. CFILL erwartet jetzt vier Parameter (Farben) auf dem Stack. HCOL wählt die Zeichenfrabe 1,2 oder 3 aus dem mit MCOL definierten Farb-Set.

Da alle Tastatur-Eingaben das Farb-RAM (high) beeinflussen, sollte man im direkten Modus nicht mit mehr als vier Farben (ändern mit MCOL) arbeiten. ?LOAD veranlaßt beim Laden der »Extras« das zusätzliche Einlesen von MPUT und MSHAPER.

Die übrigen Befehle entsprechen den normalen Hires-Befehlen. Das Laden der Multi-Color-Grafik erfolgt mit

```
FILE MAIN
3 LOAD
FCLOSE
```

und beansprucht etwa vier Minuten.

Neben diesen beiden grundsätzlichen Einstellungen findet man zusätzlich noch folgende Extras.

```
— PAINTER     (Zeichenprogramm)
— SHAPES      (der SHAPER erfordert den PAINTER)
— STRINGS     (benötigen die SHAPES)
— TURTLE      (Turtle Grafik
```

Painter

Per Tastatur-Steuerung kann ein einzelner Punkt auf dem Bildschirm bewegt werden und Linien zeichnen oder löschen. Der Aufruf erfolgt mit »x y PAINT«, worauf der »Zeichenstift« bei (x,y) erscheint. RETURN beendet den Zeichenvorgang, aber die letzten Zeichenkoordinaten verbleiben auf dem Stack. So kann man entweder mit UNPLOT den letzten Zeichenpunkt löschen oder irgendwelche Zwischenrechnungen, Farbänderungen etc. vornehmen und den PAINTER erneut mit PAINT aufrufen. Es wird immer an der letzen Stelle weitergezeichnet.

Zur Steuerung folgende Details:
- Farbeinstellung: Funktionstasten f1, f3, f5, f7
- Pen-Up/Pen-Down: G
- Bewegen des Zeichenpunktes in acht Himmelsrichtungen: R,T,Y,F,H,V,B,N

Shapes

Ein Shape ist eine x mal y Punktmatrix, wobei x ein Vielfaches von 8 ausmacht. Ein 3 x 21-Shape besteht demnach aus 24 x 21 Punkten, hat also die Größe eines Sprites. SHAPE ist eine Compiler-Erweiterung, die beliebig dimensionierte Shapes erzeugt. Der Aufruf »x y SHAPE name« definiert ein x mal y-SHAPE genannt »name«. Die Eingabe dieses Namens bewirkt dann stets den Aufruf.

```
3 21 SHAPE PETRA      (definiert PETRA)
PETRA ?S              (zeigt PETRAs Datas an)
PETRA CLEAR           (löscht PETRA)
PETRA ?S
HION
PETRA SHAPER          (zeichnet Rahmen um PETRA und
                       aktiviert den Painter)
PETRA 100 100 PUT     (zeichnet PETRA)
PETRA 110 110 PUT
HIOFF
```

Anstelle von »name« kann auch »n SPRITE« stehen, wobei n ($0 \leq n \leq 7$) sich auf das Sprite mit der Nummer n bezieht. Zuvor aber sollten alle Sprite-Pointer mit !POINTER einmal gesetzt worden sein.

Die Shapes (beziehungsweise Sprites) lassen sich mit

»name SHAPEWRITE« unter dem mit SEQ definierten Namen als sequentielles File ablegen. Ein Beispiel:

```
SEQ GIRL
PETRA SHAPEWRITE    (legt ein sequentielles File
                     namens GIRL an, dessen
                     Punktemuster PETRA entspricht)
```

Wird im Multi-Color-Modus gearbeitet, sollten Sie möglichst MPUT anstatt PUT und MSHAPER statt SHAPER verwenden.

Strings

Diese Anweisung stellt ein kleines String-Paket dar und zählt ebenfalls zu den Compiler-Erweiterungen. Strings sind wie Variable vor Benutzung zu definieren, also »name«. Ihnen stehen dann zwei Eingabevarianten frei: »name INPUT« entspricht in Basic »GETA« oder »name =$ text«, das »A$="text"« gleichkommt. Die Ausgabe erfolgt mit »name« (Arbeitsbildschirm) oder »name x y PUT$« (Grafikseite).

»n CHR« verhält sich wie ein 1 x 8-Shape, dessen Punktemuster dem Zeichen mit dem Bildschirm-Code n entspricht.

Turtle

Die Turtle-Grafik baut auf dem Hires- und Multi-Color-Paket auf und umfaßt alle Grafikbefehle des Commodore-Logo. Es beansprucht lediglich 456 Byte (!), was die Leistungsfähigkeit von Forth gut illustriert. Die Koordinaten der Turtle stehen immer als oberste Zahlen auf dem Stack (Achtung!), also die Richtung der Schildkröte in der Variablen HEADING. Bei der Übertragung von Logo-Programmen muß die UPN von Forth beachtet werden. Statt »FORWARD 10« erwartet Forth die 10 auf dem Stack, also »10 FORWARD«.

Der Punkt (0,0) liegt wie gewohnt in der linken oberen Ecke und nicht, wie bei den meisten Logo-Versionen, in der Bildschirmmitte. Dies kann der Benutzer durch »: MOVE LFORN . . .;« und entsprechende Definition von LFORN nach Bedarf ändern.

Alle Turtle-Befehle können natürlich wie bei Logo abgekürzt werden. Sie schreiben dann statt »10 FORWARD« einfach »10 FD« und so fort. Die Turtle sollte sich nicht mehr als 30000 Punkte in jeder Richtung vorwärts bewegen (2-Byte-Arithmetik). Der Befehl TURTLE initialisiert die Schildkröte und das Farb-RAM, löscht die Grafikseite und sollte nur dann aufgerufen werden, wenn man sich das HION ersparen will – also nicht von der Grafikseite aus. Ein Beispielprogramm:

```
: LINIE 50 FD 90 RT ;
: QUADRAT LINIE LINIE LINIE LINIE
: ROSETTE 36 0 DO QUADRAT 10 RT 12 FD LOOP ;

TURTLE ROSETTE
```

Soweit der allgemeine Überblick über die einzelnen, im Turtle-Forth enthaltenen Programm-Pakete.

Die Screens von Turtle-Forth

SCR # 1: nicht verwendet.

SCR # 2 bis 4: Lade-Screens. SCR # 2 lädt die normale Hires-Grafik, wobei die Befehle PLOT und LINK wahlweise als Primitive (SCR # 70 bis 77) oder als High-Level (SCR # 12 bis 13 und 20) definiert werden – je nachdem wie die Klammern in SCR # 2 stehen.

SCR # 3 lädt die Multi-Color-Grafik, bei der sowohl der normale als auch der Multi-Color-Modus zur Auswahl stehen kann. Die Umschaltung zwischen den beiden Modi erfolgt Mit den Wörtern MON beziehungsweise MOFF. Zu beachten ist, daß der Dictionary-Pointer zwischendurch auf 16384 hochgesetzt wird, um 8 KByte Platz für die Bitmap zu schaffen (siehe rechts). SCR # 4 lädt die Extras, die natürlich auch einzeln zu verwenden sind.

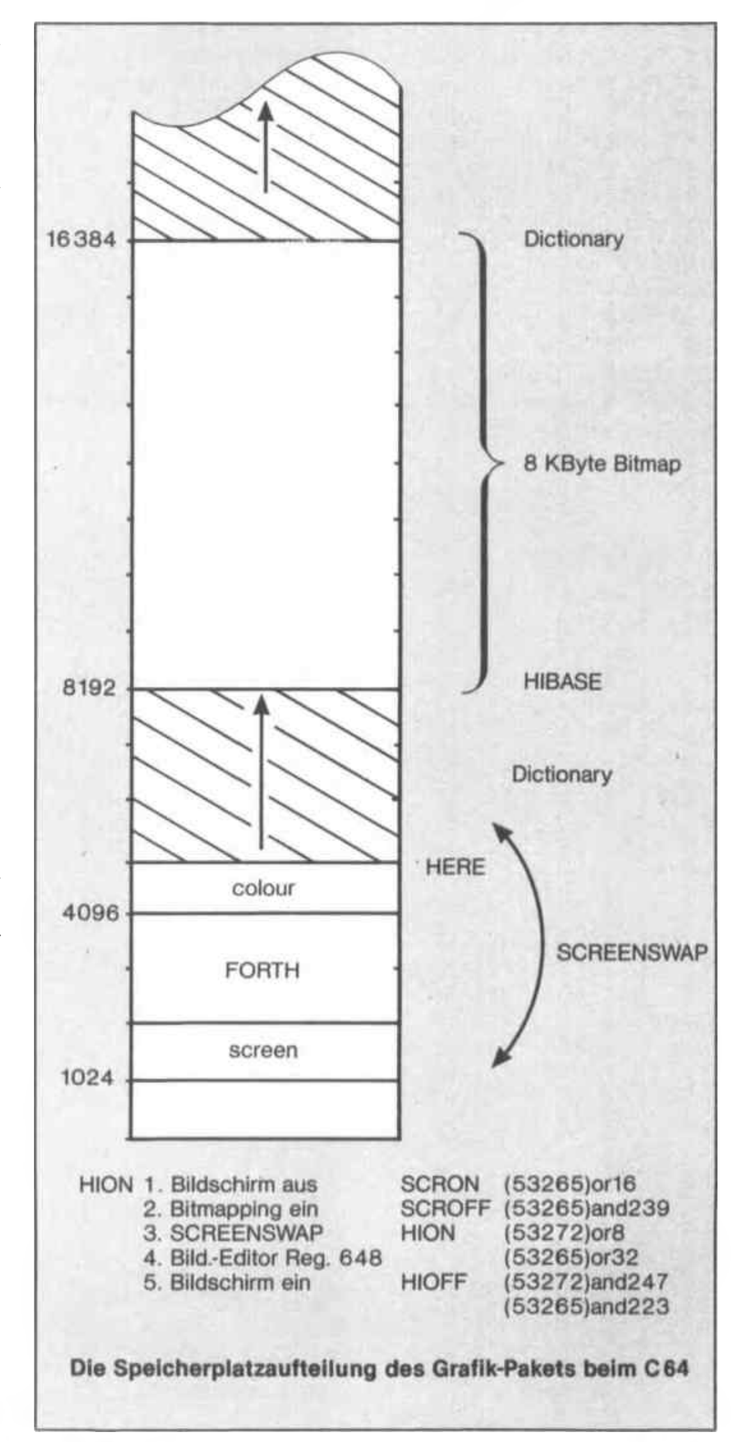
SCR # 5 bis 9: nicht verwendet.

SCR # 10 bis 11: Diese beiden Screens enthalten die Installation der hochauflösenden Grafik und müssen dem Computertyp angepaßt werden. Die hier vorgestellten Wörter gelten für den Commodore 64.

Zunächst werden 1000 Byte Platz geschaffen, um den Bildschirm zwischenzuspeichern. Das bezweckt, daß Sie auch bei eingeschalteter Hires-Grafik weiterhin alle Befehle im Direktmodus eingeben können. SCREENSWAP besorgt die Speicherverschiebung und benutzt das Wort NSWAP (adr 1 adr 2 n –), welches n Byte zwischen den Adressen adr_1 und adr_2 austauscht. SCRON beziehungsweise SCROFF schalten den Videochip ein und aus, um einen sauberen Übergang beim Einschalten der Grafik zu erzielen.

HION und HIOFF besorgen die Umschaltung auf hochauflösende Grafik. Falls Sie mit einem anderen Computer als dem C64 arbeiten, müssen hier natürlich andere »Pokes« stehen, die Sie dem Benutzerhandbuch entnehmen können.

Falls die Hires eingeschaltet ist, legt HION? ein Flag (ungleich Null) auf den Stack, ansonsten eine Null. SCREEN teilt dem Bildschirmeditor des Betriebssystems mit, wo der



Die Speicherplatzaufteilung des Grafik-Pakets beim C64

Eingabe-Bildschirm liegt. Bei ausgeschalteter Hires befindet er sich ab Register 1024 (also Page 4), bei eingeschalteter Hires, also nach SCREENSWAP ab Register 4096 (also Page 16).

Bei Forth-Programmen empfiehlt es sich, Wort für Wort einzugeben und auszutesten. Dann weiß man nämlich sicher, wo man steht und spart sich viel Ärger beim Fehlersuchen. Sie kennen nun alle Wörter, um mit der hochauflösenden Grafik auch umzugehen.

SCR # 12 bis 13: Mit PLOT und UNPLOT setzen oder löschen Sie die einzelnen Punkte. -PLOT bewirkt PLOT oder UNPLOT je nach Verhältnis der Variablen VPLOT und VBREAK. Es wird von LINK benutzt. Dadurch ist auch UNLINK einfach als Spezialfall von LINK zu definieren und zudem lassen sich alle Linien auch unterbrochen (BREAK) darstellen.

ADR bewirkt die Adreß-Umrechnung von X/Y-Koordinaten in die Registeradresse der Bitmap.

SCR # 14 bis 15: Bevor wir mit der Hires (CIRCLE) fortfahren, wollen wir die Multi-Color-Grafik installieren. SCR # 14 und 15 entsprechen bis auf kleine Details SCR # 10 und 11.

SCR # 16 bis 19: Hier wirkt eine trickreiche Programmier-Technik. Damit sowohl die Arbeit mit normaler Hires als auch mit Multi-Color-Hires möglich ist, sind die meisten Wörter zweimal zu definieren. Um den gleichen Namen wieder zu verwenden, werden die Wörter vektoriell ausgeführt.

Zum Beispiel PLOT. Es führt das Wort aus (EXECUTE), das in der Variablen 'PLOT steht. Dort befindet sich im normalen Hires-Modus die Code-Feld-Adresse (CFA) von PLOT,H im Multi-Color-Modus dagegen die CFA von PLOT,M. Das Einbeziehungsweise Ausschalten mit MON und MOFF ändert also lediglich die Vektoren 'PLOT 'ADR etc. ab. MCON und MCOFF bewirken dabei die eigentlichen »Pokes«, um den entsprechende Grafik-Modus einzuschalten.

SCR # 20: Auf dem PLOT-Befehl aufbauend zeichnen wir jetzt eine Gerade, das heißt, alle Punkte zwischen zwei Koordinaten. LINK verwendet, je nachdem, ob die Steigung der Geraden mehr als 45 Grad beträgt, LINX oder LINY. Nun gibt auch BREAK einen einleuchtenden Sinn.

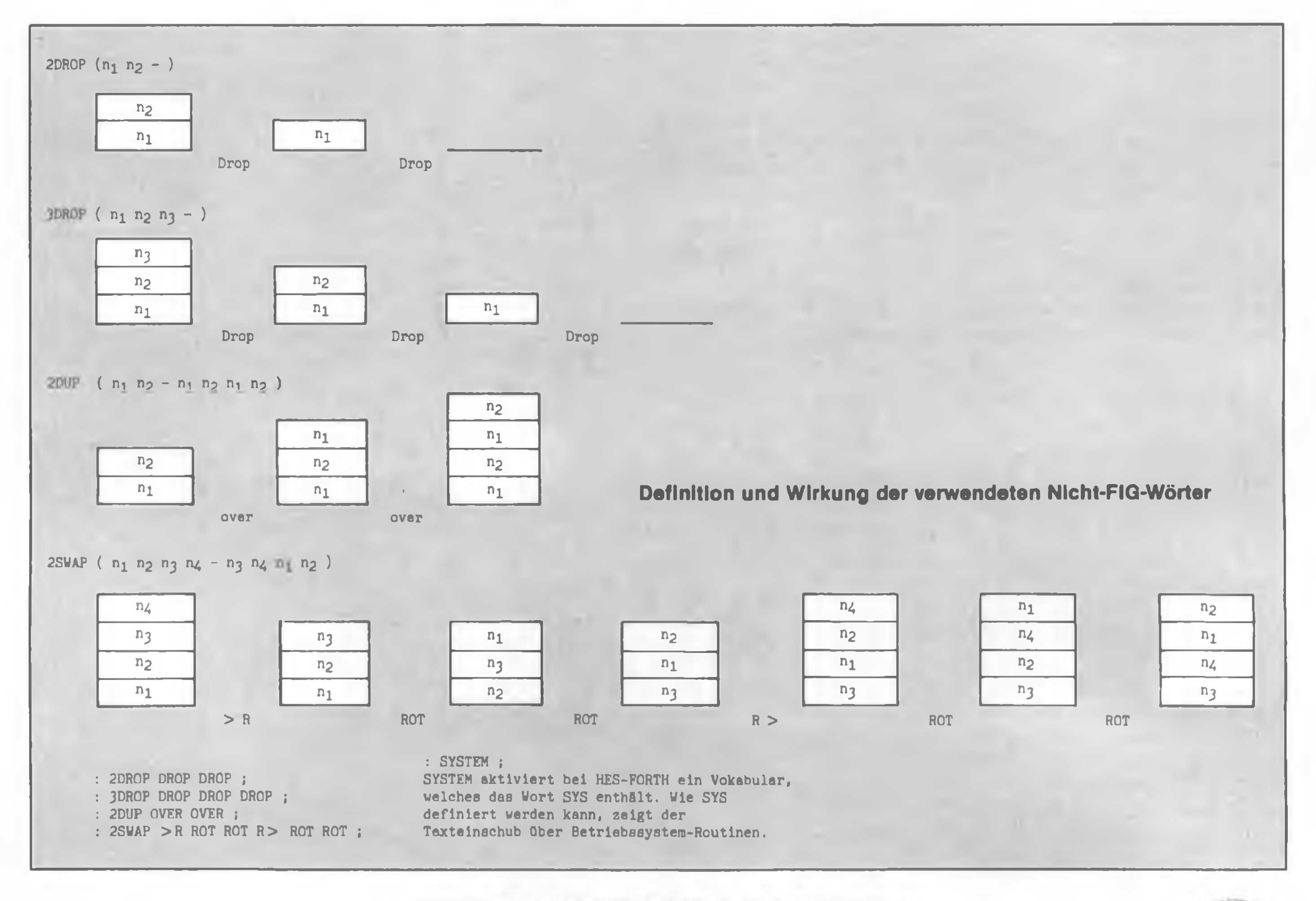
SCR # 21 bis 23: Um einen Kreis zu zeichnen, benötigen wir eine Sinus- oder Cosinus-Funktion. Das scheint schwierig, da Forth doch keine Fließkommazahlen kennt.

Wir erzeugen uns jedoch einfach Grad für Grad eine Liste der Sinuswerte. (Wegen der Beziehung cos α = sin (90-α) kann man sich eine Cosinus-Liste sparen.) Das entscheidende Wort ist nun CIRC. Es berechnet aus dem Winkel a die Koordinaten x und y unter Benutzung des Radius, der in der Variablen RAD stehen soll. ARC zeichnet dann einen Kreisbogen, wobei die mit CIRC gewonnenen Koordinaten mit dem Quotienten EPS/100 multipliziert werden, so daß sich die Kreise stauchen und dehnen lassen. EPS ist zu 100 initialisiert. Es entstehen also wirklich Kreise, solange man die Exzentrität nicht mit EXCENTER ändert. CIRCLE ist dann nur noch ein Spezialfall von ARC.

SCR # 24: SCR # 24 demonstriert schön die Forth-Philosophie: NGON zeichnet offene, POLYGON geschlossene Kurvenzüge. Beide benutzen dazu ein Wort namens GON. TRIC und REC, also Drei- und Vierecke – Spezialfälle geschlossener Kurven – arbeiten also mit POLYGON. QUADRAT wiederum ist ein Sonderfall von REC.

SCR # 25 bis 27: Hier wird das Wort SYS (a x y adr – a x y) benötigt, das nicht dem FIG-Wortschatz entstammt. Die FIG-Definition eines solchen Wortes findet sich im Texteinschub »Einbinden von Betriebssystem-Routinen«. SF ist eine Stringvariable, die mit SEQ eingelesen und mit .SEQ ausgegeben wird.

SWRITE ruft einige Betriebssystem-Routinen zur sequentiellen Datenspeicherung auf:



Definition und Wirkung der verwendeten Nicht-FIG-Wörter

```
$ FFBD  Setnam
$ FFBA  Setlfs
$ FFCO  Open
$ FFC9  CHKOUT
```

SREAD ruft auf:

```
$ FFBD  Setnam
$ FFBA  Setlfs
$ FFCO  Open
$ FFC6  CHKIN
```

SCLOSE ruft auf:

```
$ FFC3  Close
$ FFCC  Clrchn
```

So lassen sich einzelne Bytes (TO FROM) und ganze Speicherbereiche (TOS FROMS) auf Diskette sequentiell speichern. Diese Routinen sind Commodore-spezifisch und sind auf anderen Computern unbrauchbar. Die Befehle sind aber sicherlich für viele C 64-Forth-Programme äußerst nützlich.

SCR # 28: HIWRITE und HIREAD benutzen nun die Fähigkeit zu sequentieller Datenspeicherung, um Grafiken zu speichern und wiederzulesen. Folgende Adreßbereiche können bearbeitet werden.:

1. 8 KByte Bitmap (8192 bis 16383)
2. Hintergrund (53281)
3. Bildschirm (1024 bis 2023 beziehungsweise 4096 bis 5095)
4. Farbspeicher (55296 bis 56295)

SCR # 29: nicht verwendet.

SCR # 30 bis 32: Painter-Package (tastaturgesteuertes Zeichenprogramm).

SCR # 33 bis 39: nicht verwendet.

SCR # 40 bis 42: Shape-Package (Teil 1).

SCR # 43 bis 44: nicht verwendet.

SCR # 45 bis 46: Shape-Package (Teil 2).

SCR # 47 bis 49: nicht verwendet.

SCR # 50: String-Package, grundsätzliche Definitionen.

SCR # 51 bis 54: nicht verwendet.

SCR # 55 bis 57: String-Package, zusätzliche Funktionen.

SCR # 58 bis 59: nicht verwendet.

SCR # 60 bis 61: Turtle-Package, stellt in nur zwei Screens alle für die Turtle-Grafik notwendigen Befehle zusammen.

SCR # 62 bis 69: nicht verwendet.

SCR # 70 bis 77: Nun lernen Sie die Wörter PLOT und LINK auch als Primitive (für C 64) kennen. Dazu benötigen Sie das Assembler-Vokabular Ihres FIG-Forth und ein ganzes Stück Arbeit, um die Screens einzugeben. Dafür haben Sie dann aber auch einen sehr schnellen LINK-Algorithmus.

XFORM und YFORM passen die Koordinaten an, die »außerhalb« liegen, das heißt, die Werte $0 \geq x \geq 319$ oder $0 \geq y \geq 199$, so daß Linien, die auf einer Seite herauslaufen, auf der anderen wieder auftauchen. (»Wrap-Around«.) Der fortgeschrittene Programmierer kann sich diese Wörter auch so abändern, daß die Linie am Rand einfach abbricht, also einen maximalen Wert nicht überschreitet (»Clipping«). Bemerkenswert ist vielleicht, daß UNLINK den Code von LINK abändert, indem es die Adresse von UNPLOTCODE in den Code-Body von PLOTIT schreibt und dann einfach LINK aufruft. Dies ist eine Programmiertechnik, die sehr mit Vorsicht zu genießen ist!

Eine Liste aller Befehle mit ihren Wirkungen ist in der Tabelle (rechts) enthalten.

Aufbauend auf dem Grundprogramm können Sie eine Vielzahl eigener Programme verwirklichen. Als Anregung einige Beispiele, wie das tastaturgesteuerte Zeichenprogramm (PAINTER), das Speichern und Kopieren von beliebigen Bildschirmausschnitten (SHAPES), der Sprite-Generator (SHAPER) oder die Logo entlehnte Schildkröte (TURTLE). Schließlich können Sie auch Texte (STRINGs) aus dem Character-ROM auslesen und in die HiRes einbringen.

(Andreas Carl/ev)

(1) HiRes

Befehl	Stack	Wirkung
HIBASE	(- 8192)	Adresse des ersten Bytes
SCROFF	(-)	ausschalten des Video-Chip
SCRON	(-)	einschalten des Video-Chip
HION	(-)	umschalten auf Grafikbildschirm
HIOFF	(-)	umschalten auf Arbeitsbildschirm
HION?	(-f)	Grafikbildschirm eingeschaltet?
HFILL	(n-)	füllt die Grafikseite mit dem Byte n
HCLEAR	(-)	löscht den Grafikbildschirm
HCOL	(n-)	wählt Zeichenfarbe $0 \leq n \leq 15$
CFILL	(b c -)	füllt den Farbspeicher mit der Hintergrundfarbe b und der Zeichenfarbe c
ADR	(x y - a)	Wandelt die Koordinaten x, y in die Bitmap-Adresse a um
PLOT	(x y -)	setzt Zeichenpunkt
UNPLOT	(x y -)	löscht Zeichenpunkt
?PLOT	(x y - f)	Zeichenpunkt gesetzt?
XFORM	:;	Handler für Koordinaten, die »außerhalb« liegen;
LINK	(x_0y_0 x_1y_1 -)	verbindet zwei Punkte P_0 und P_1
UNLINK	(x_0y_0 x_1y_1 -)	löscht die Verbindungslinie zweier Punkte P_0 und P_1
BREAK	(n -)	definiert die Länge der Teile von gebrochenen Linien. Ist zu n = 30000 initialisiert
NGON	(x_0y_0..x_{n-1}..y_{n-1} n -)	verbindet n Punkte P_0 bis P_{n-1}
POLYGON	(x_0y_0..$x_{n-1}y_{n-1}$ n -)	verbindet n Punkte P_0 bis P_{n-1} zu einem geschlossenen Kurvenzug
TRIC	(x_0y_0 x_1y_1 x_2y_2 -)	zeichnet ein Dreieck $P_0P_1P_2$
REC	(x_0y_0 ab—)	zeichnet ein Rechteck mit der Länge und der Höhe b
QUADRAT	(x_0y_0 a -)	zeichnet ein Quadrat mit der Kantenlänge a
CIRCLE	(x_0y_0 r d α -)	zeichnet einen Kreis um den Mittelpunkt P_0 mit dem Radius r in Schritten von je d α Grad
ARC	(x_0y_0 r α_0 α_1 dα-)	zeichnet einen Teilkreis von α_0 Grad bis α_1 Grad
ANGL	x_0y_0 r α -)	zeichnet Radius eines Kreises zum Winkel α
EXCENTER	(n -)	bestimmt die Exzentrizität für CIRCLE, ARC und ANGL; n= 100 bedeutet $r_x : r_y = 1$
SEQ name	(-)	legt Filenamen für folgende Diskettenoperationen fest
.SEQ	(-)	zeigt Filenamen an
SWRITE	(-)	öffnet sequentielles Schreibfile
SREAD	(-)	öffnet sequentielles Lesefile
SCLOSE	(-)	schließt sequentielles File
TO	(-)	schreibt Byte b in offenes File
TOS	(a n -)	schreibt n Bytes ab der Adresse a in offenes File
FROM	(- b)	liest Byte b ein
FROMS	(a n -)	liest n Bytes ein und speichert sie ab der Adresse a
HIWRITE	(-)	schreibt 8 KByte HiRes plus 3 KByte Farbinformation in sequentielles File
HIREAD	(-)	liest HiResgrafik aus sequentiellem File ein
SHAPEWRITE	(-)	speichert Shapes oder Sprites in sequentielles File
SHAPEREAD	(-)	liest Shape oder Sprite

(2) Multi-Color-Grafik

alle Befehle wie (1) und zusätzlich:

Befehl	Stack	Wirkung
MON	(-)	schaltet auf Multi-Color-Mode um
MOFF	(-)	schaltet auf Normal-Mode um
MFLAG	(- f)	Multi-Color-Mode eingeschaltet?
CFILL	(b $c_1c_2c_3$ -)	füllt die Farbspeicher
MCOL	($c_1c_2c_3$ -)	wählt die drei Zeichenfarben ($0 \leq c \leq 15$)
HCOL	(n -)	wählt aktuelle Farbe für PLOT etc. (n = 1,2,3)

Die Befehle des Grafik-Pakets

Listing »Turtle-Grafik« in Forth (Fortsetzung)

```
SCR # 18
 0          ( CONT. )
 1    ' ADR,H     CFA VARIABLE 'ADR
 2    ' ?PLOT,H   CFA VARIABLE '?PLOT
 3    ' PLOT,H    CFA VARIABLE 'PLOT
 4    ' UNPLOT,H  CFA VARIABLE 'UNPLOT
 5    ' HCOL,H    CFA VARIABLE 'HCOL
 6    ' CFILL,H   CFA VARIABLE 'CFILL
 7
 8 : ADR 'ADR @ EXECUTE ;
 9 : ?PLOT '?PLOT @ EXECUTE ;
10 : PLOT 'PLOT @ EXECUTE ;
11 : UNPLOT 'UNPLOT @ EXECUTE ;
12 : -PLOT VPLOT @ VBREAK @ / 2 MOD
13          0= IF PLOT ELSE UNPLOT ENDIF VPLOT DUP @ 1+ SWAP ! ;
14 : HCOL 'HCOL @ EXECUTE ;
15 : CFILL 'CFILL @ EXECUTE ;                               -->

SCR # 19
 0          ( CONT. )
 1
 2
 3 : MCON 53270 C@ 16 OR 53270 C! ;
 4 : MCOFF 53270 C@ 239 AND 53270 C! ;
 5
 6 : MON MCON ' PLOT,M CFA 'PLOT ! ' UNPLOT,M CFA 'UNPLOT !
 7            ' HCOL,M CFA 'HCOL ! ' CFILL,M  CFA 'CFILL  !
 8            ' ADR,M  CFA 'ADR  ! ' ?PLOT,M  CFA '?PLOT  !
 9           1 ' MFLAG ! ;
10 : MOFF MCOFF ' PLOT,H CFA 'PLOT ! ' UNPLOT,H CFA 'UNPLOT !
11              ' HCOL,H CFA 'HCOL ! ' CFILL,H  CFA 'CFILL  !
12              ' ADR,H  CFA 'ADR  ! ' ?PLOT,H  CFA '?PLOT  !
13           0 ' MFLAG ! ;
14
15

SCR # 20
 0          ( LINK )
 1
 2     0 VARIABLE DX    0 VARIABLE DY    0 VARIABLE NSTEP
 3
 4 : TO   DUP 0< IF 1 - -1 NSTEP ! ELSE 1 + 1 NSTEP ! ENDIF ;
 5
 6 : LINX DO DUP I DY @ DX @ */ +
 7         ROT DUP I + ROT -PLOT SWAP NSTEP @ +LOOP 2DROP ;
 8 : LINY DO DUP I DX @ DY @ */ +
 9         ROT DUP I + ROT SWAP -PLOT SWAP NSTEP @ +LOOP 2DROP ;
10
11 : LINK INIPLOT ROT >R I - DY ! SWAP >R I - DX !
12         R> R> DX @ ABS DY @ ABS > IF DX @ TO 0 LINX
13                         ELSE SWAP DY @ TO 0 LINY ENDIF ;
14 : UNLINK VBREAK @ >R 0 VBREAK ! LINK R> VBREAK ! ;
15

SCR # 21
 0          ( SINUS )
 1 0 VARIABLE SINA 175 , 349 , 523 , 698 , 872 , 1045 , 1219 , 1392
 2  , 1564 , 1763 , 1908 , 2079 , 2250 , 2419 , 2588 , 2756 , 2924
 3  , 3090 , 3256 , 3420 , 3584 , 3746 , 3907 , 4067 , 4226 , 4384
 4  , 4540 , 4695 , 4848 , 5000 , 5150 , 5299 , 5446 , 5592 , 5736
 5  , 5878 , 6018 , 6157 , 6293 , 6428 , 6561 , 6691 , 6820 , 6947
 6  , 7071 , 7193 , 7314 , 7431 , 7547 , 7660 , 7771 , 7880 , 7986
 7  , 8090 , 8192 , 8290 , 8387 , 8480 , 8572 , 8660 , 8746 , 8829
 8  , 8910 , 8988 , 9063 , 9135 , 9205 , 9272 , 9336 , 9397 , 9455
 9  , 9511 , 9563 , 9613 , 9659 , 9703 , 9744 , 9781 , 9816 , 9848
10  , 9877 , 9903 , 9925 , 9945 , 9962 , 9976 , 9986 , 9994 , 9998
11  , 10000 ,
12
13 : SIN@ 2 * SINA + @ 10000 SWAP ;
14 : COS@ 90 SWAP - SIN@ ;
15

SCR # 22
 0          ( CIRCLE )
 1
 2     0 VARIABLE DA    0 VARIABLE RAD    100 VARIABLE EXC
 3
 4 : CNORM BEGIN DUP   0 < WHILE 360 + REPEAT
 5         BEGIN DUP 360 > WHILE 360 - REPEAT ;
 6
 7 : SIN CNORM DUP DUP ABS / SWAP ABS DUP 90 < IF SIN@ ELSE
 8       DUP 180 < IF 180 SWAP - SIN@ ELSE
 9       DUP 270 < IF 180 - SIN@ -1 * ELSE
10       360 SWAP - SIN@ -1 * ENDIF ENDIF ENDIF ROT * ;
11 : COS 90 SWAP - SIN ;
12
13 : EPS EXC @ 100 */ ;
14 : EXCENTER EXC ! ;                                        -->
15

SCR # 23
 0          ( CONT. )
 1
 2   0 VARIABLE CX     0 VARIABLE CY
 3
 4 : CIRC RAD @ OVER SIN SWAP */ RAD @ ROT COS SWAP */ ;
 5
 6 : ARC DA ! 2DUP > IF DA DUP @ -1 * SWAP ! ENDIF
 7        >R >R RAD ! CY ! CX ! R> R> SWAP
 8        DO I CIRC CX @ + SWAP EPS CY @ + R> DA @ + >R
 9           I CIRC CX @ + SWAP EPS CY @ + R> DA @ - >R
10           LINK DA @ +LOOP ;
11 : CIRCLE 0 360 ROT ARC ;
12 : ANGL   >R RAD ! 2DUP R> CIRC >R + SWAP R> + SWAP LINK ;
13
14
15

SCR # 24
 0          ( POLYGON )
 1
 2   0 VARIABLE B     0 VARIABLE L
 3
 4 : GON      1 DO >R >R 2DUP R> R> 2SWAP LINK LOOP ;
 5 : NGON     GON 2DROP ;
 6 : POLYGON ROT ROT >R DUP I SWAP >R ROT GON R> R> LINK ;
 7 : TRIC 3 POLYGON ;
 8 : REC  1 - B ! 1 - L !
 9         2DUP L @ ROT + SWAP
10         2DUP B @ +
11         2DUP L @ ROT SWAP - SWAP
12         2DUP B @ -
13         5 NGON ;
14 : QUADRAT DUP REC ;
15

SCR # 25
 0          ( SEQUENTIELLE FILES )
 1
 2 0 VARIABLE SF 20 ALLOT
 3
 4 :  SEQ 34 WORD HERE SF 20 CMOVE ;
 5 : .SEQ SF COUNT TYPE ;
 6
 7 : NAM,S,X SF COUNT + DUP ASCII , SWAP C!
 8                      1+ DUP ASCII S SWAP C!
 9                      1+ DUP ASCII , SWAP C!
10                      1+                  C!
11                 SF DUP C@ 4 + SWAP C! ;
12 : NAM,S,W ASCII W NAM,S,X ;
13 : NAM,S,R ASCII R NAM,S,X ;
14                                                          -->
15

SCR # 26
 0          ( CONT. )
 1
 2   HEX   0 VARIABLE SPSAVE
 3
 4 : OFFSPRITE D015 C@ SPSAVE C! 0 D015 C! ;
 5 : OLDSPRITE SPSAVE C@ D015 C! ;
 6
 7 : SWRITE OFFSPRITE CR ." WRITING " .SEQ
 8          SYSTEM NAM,S,W SF COUNT SWAP 100 /MOD FFBD SYS
 9          3DROP SF DUP C@ 4 - SWAP C!
10          2 8 2 FFBA SYS FFC0 SYS 3DROP
11          0 2 0 FFC9 SYS 3DROP ;
12
13 : TO SYSTEM 0 0 FFD2 SYS 3DROP ;
14 : TOS 0 DO DUP I + C@ TO LOOP DROP ;                     -->
15

SCR # 27
 0          ( CONT. )
 1
 2
 3 : SREAD OFFSPRITE CR ." READING " .SEQ
 4         SYSTEM NAM,S,R SF COUNT SWAP 100 /MOD FFBD SYS
 5         3DROP SF DUP C@ 4 - SWAP C!
 6         2 8 2 FFBA SYS FFC0 SYS 3DROP
 7         0 2 0 FFC6 SYS 3DROP ;
 8
 9 : SCLOSE OLDSPRITE SYSTEM 2 0 0 FFC3 SYS FFCC SYS 3DROP ;
10
11 : FROM SYSTEM 0 0 0 FFCF SYS 2DROP ;
12 : FROMS 0 DO FROM OVER I + C! LOOP DROP ;
13
14                                   DECIMAL
15

SCR # 28
 0          ( WRITE/READ )
 1
 2
 3 : HIWRITE SWRITE 8192 DUP TOS 53281 C@ TO
 4                  HION? IF 1024 ELSE 4096 ENDIF 1000 TOS
 5                  55296 1000 TOS SCLOSE ;
 6 : HIREAD SREAD 8192 DUP FROMS FROM 53281 C!
 7                  HION? IF 1024 ELSE 4096 ENDIF 1000 FROMS
 8                  55296 1000 FROMS SCLOSE ;
 9
10 : SHAPEWRITE SWRITE * TOS   SCLOSE ;
11 : SHAPEREAD SREAD  * FROMS SCLOSE ;
12
13
14
15
```

```
SCR # 30
 0          ( PAINTER )
 1   0 VARIABLE OLD
 2 : REPON 128 650 C! ;
 3 : REPOFF 0 650 C! ;
 4 : KILL 2DUP UNPLOT ;
 5 : PLOTOVER 2DUP ADR OLD @ SWAP C! ;
 6 : DRAW ;
 7   ' DRAW     CFA VARIABLE LASTVECTOR
 8   ' PLOTOVER CFA VARIABLE PENMODE
 9 : UP/DOWN PENMODE @ LASTVECTOR @ PENMODE ! LASTVECTOR ! ;
10 : ?DOWN ' PLOTOVER CFA PENMODE @ = ;
11 : SETMODUS ?DOWN IF LASTVECTOR ! ELSE PENMODE ! ENDIF ;
12 : DODRAW  ' DRAW  CFA SETMODUS ;
13 : DOKILL ' KILL CFA SETMODUS ;
14 : LASTPLOT >R LASTVECTOR @ EXECUTE R> ;
15 : NEXTPLOT 2DUP ADR C@ OLD ! 2DUP PLOT ;                  -->

SCR # 31
 0          ( CONT. )
 1 : SLOPE BEGIN KEY DUP 13 = 0= WHILE LASTPLOT
 2         DUP 72 = IF >R SWAP 1+ SWAP R> ENDIF
 3         DUP 78 = IF >R 1+ SWAP 1+ SWAP R> ENDIF
 4         DUP 66 = IF >R 1+ R> ENDIF
 5         DUP 86 = IF >R 1+ SWAP 1 - SWAP R> ENDIF
 6         DUP 70 = IF >R SWAP 1 - SWAP R> ENDIF
 7         DUP 82 = IF >R 1 - SWAP 1 - SWAP R> ENDIF
 8         DUP 84 = IF >R 1 - R> ENDIF
 9         DUP 89 = IF >R 1 - SWAP 1+ SWAP R> ENDIF
10        DUP  71 = IF UP/DOWN ENDIF
11        DUP 133 = IF 3 HCOL DODRAW ENDIF
12        DUP 134 = IF 2 HCOL DODRAW ENDIF
13        DUP 135 = IF 0 HCOL DODRAW ENDIF
14            136 = IF DOKILL ENDIF
15        NEXTPLOT REPEAT DROP ;                  -->

SCR # 32
 0          ( CONT. )
 1
 2 : PAINT 2DUP PLOT
 3         ' DRAW     CFA LASTVECTOR !
 4         ' PLOTOVER CFA PENMODE !
 5         REPON SLOPE REPOFF ;
 6
 7
 8   (      R T Y                  F1
 9                                 F3
10          F G H                  F5
11                                 F7
12           V B N    RETURN             )
13
14
15

SCR # 40
 0          ( SHAPES )
 1
 2   0 VARIABLE PARITY
 3
 4 : SHAPE <BUILDS OVER , DUP , * ALLOT
 5         DOES> >R R 4 + R @ R> 2+ @ ;
 6 : CLEAR * 0 DO 0 OVER I + C! LOOP DROP ;
 7 : ?S OVER HERE ! * 0 DO I HERE @ MOD 0=
 8      IF CR ENDIF DUP I + C@ 4 .R LOOP DROP ;
 9 : DIM ROT DROP ;
10 : ZAHL? 0 IN ! TIB @ 10 EXPECT
11         32 WORD HERE NUMBER DROP ;
12 : INPUT OVER PARITY ! * 0 DO I PARITY @ MOD 0= IF CR ENDIF
13                             ZAHL? OVER I + C! LOOP DROP ;
14
15                                                        -->

SCR # 41
 0          ( CONT. )
 1    0 VARIABLE SX    0 VARIABLE SY
 2
 3 : >BIT 8 0 DO 2 * >R I 256 / R> 255 AND LOOP DROP ;
 4 : 8SWAP 8 0 DO HERE I + C! LOOP
 5         8 0 DO HERE I + C@ LOOP ;
 6 : PUT SY ! SX ! DY ! DX !  0 SWAP
 7      DX @ DY @ * 0 DO DUP I + C@ >BIT 8SWAP
 8                        8 0 DO SX @ SY @ ROT
 9                               IF PLOT ELSE UNPLOT ENDIF
10                               1 SX +! LOOP
11                        >R 1+ DUP DX @ MOD 0= IF 1 SY +!
12                                                 -8 DX @ * SX +!
13                                             ENDIF R> LOOP 2DROP ;
14
15                                                   45 ?LOAD

SCR # 42
 0          ( SHAPER )
 1   16 VARIABLE PX   16 VARIABLE PY
 2
 3 : 3DUP >R OVER OVER R ROT ROT R> ;
 4 : FRAME ROT DROP 2+ SWAP 8 * 2+ SWAP 15 15 2SWAP REC ;
 5 : DEFSHAPE OVER HERE ! * 16 PX ! 15 PY !
 6           0 DO I HERE @ MOD 0= IF 1 PY +! 16 PX !
 7                                  ELSE 8 PX +! ENDIF
 8                PX @ PY @ ADR C@ OVER C! 1+ LOOP DROP ;
 9 : SHAPER HCLEAR 3DUP FRAME 17 17 PAINT
10          2DROP DEFSHAPE ;
11
12 : SPRITE 2040 + C@ 64 * 3 21 ;
13 : !POINTER 8 0 DO 48 I + 2040 I + C! LOOP ;
14
15                                                  46 ?LOAD

SCR # 45
 0          ( MPUT )
 1
 2 : MPUT SY ! SX ! DY ! DX !  0 SWAP
 3        DX @ DY @ * 0 DO DUP I + C@ >BIT 8SWAP
 4                         4 0 DO 2 * * DUP
 5                                IF HCOL SX @ SY @ PLOT
 6                                ELSE DROP SX @ SY @ UNPLOT ENDIF
 7                                1 SX +! LOOP
 8                         >R 1+ DUP DX @ MOD 0= IF 1 SY +!
 9                                                  -4 DX @ * SX +!
10                                              ENDIF R> LOOP 2DROP ;
11
12
13
14
15

SCR # 46
 0          ( MSHAPER )
 1
 2
 3 : MFRAME ROT DROP 2+ SWAP 4 * 2+ SWAP 7 15 2SWAP REC ;
 4 : DEFMSHAPE OVER HERE ! * 16 PX ! 15 PY !
 5           0 DO I HERE @ MOD 0= IF 1 PY +! 16 PX !
 6                                  ELSE 8 PX +! ENDIF
 7                PX @ PY @ ADR,H C@ OVER C! 1+ LOOP DROP ;
 8
 9 : MSHAPER HCLEAR 3DUP MFRAME 9 17 PAINT
10          2DROP DEFMSHAPE ;
11
12
13
14
15

SCR # 50
 0          ( STRING-PACKAGE )
 1
 2 :       $ <BUILDS 78 ALLOT DOES> ;
 3
 4 : INPUT$ 0 IN ! TIB @ 80 EXPECT 34 WORD
 5         HERE SWAP 80 CMOVE ;
 6 :     =$ 34 WORD HERE SWAP 80 CMOVE ; IMMEDIATE
 7
 8 :      .$ COUNT TYPE ;
 9 :    LEN COUNT SWAP DROP ;
10
11
12
13
14
15

SCR # 55
 0          ( @CHAR )
 1
 2 CODE @CHAR 254 # LDA, 56334 AND, 56334 STA,
 3            251 # LDA, 1 AND, 1 STA,
 4
 5            BOT LDA, N 1 - STA,
 6            BOT 1+ LDA, N STA,
 7            0 # LDA, BOT 1+ STA, TAY,
 8            N 1 - )Y LDA, BOT STA,
 9
10            4 # LDA, 1 ORA, 1 STA,
11            1 # LDA, 56334 ORA, 56334 STA,
12
13            NEXT JMP, END-CODE
14
15

SCR # 56
 0          ( ASCII-WANDLER )
 1
 2
 3 : A>S DUP 128 AND IF 127 AND 64 OR ELSE
 4      DUP  64 AND 0= IF ELSE
 5      DUP  32 AND IF 95 AND ELSE
 6           63 AND ENDIF ENDIF ENDIF ;
 7
 8
 9
10
11
12
13
14
15
```



```
SCR # 57
 0         ( PUT8 )
 1
 2     53248 CONSTANT CHARBASE
 3         0 VARIABLE CHARACTER 6 ALLOT
 4
 5 : CHR 8 * CHARBASE +
 6       8 0 DO DUP I + @CHAR CHARACTER I + C!
 7           LOOP DROP CHARACTER 1 8 ;
 8
 9         0 VARIABLE CHX    0 VARIABLE CHY
10
11 : PUT8 CHY ! CHX ! COUNT 0 DO DUP I + C@ A>S CHR
12                              CHX @ I 8 * + CHY @ PUT
13                           LOOP DROP ;
14
15

SCR # 60
 0         ( TURTLE )
 1
 2     0 VARIABLE HEADING     1 VARIABLE PENSTATE
 3
 4
 5 : RIGHT HEADING +! ;
 6 : LEFT  -1 * RIGHT ;
 7 : MOVE 2DUP >R >R 2SWAP R> R>
 8        PENSTATE @ IF LINK ELSE 2DROP 2DROP ENDIF ;
 9 : FORWARD RAD ! 2DUP HEADING @ 90 - CIRC >R + SWAP R> +
10          SWAP MOVE ;
11 : BACK 180 RIGHT FORWARD 180 LEFT ;
12
13
14                                                    -->
15

SCR # 61
 0         ( CONT. )
 1
 2 : PENDOWN 1 PENSTATE ! ;
 3 : PENUP 0 PENSTATE ! ;
 4 : SETHEADING HEADING ! ;
 5 : SETX OVER MOVE ;
 6 : SETY >R OVER R> MOVE ;
 7 : SETXY MOVE ;
 8 : HOME 0 SETHEADING 160 100 MOVE ;
 9 : TURTLE HCLEAR HION 0 SETHEADING MFLAG IF 11 1 0 6 CFILL
10          80 100 ELSE 11 1 CFILL 160 100 ENDIF ;
11
12   : FD FORWARD ;          : BK BACK ;
13   : RT RIGHT ;            : LT LEFT ;
14   : PD PENDOWN ;          : PU PENUP ;
15   : SETH SETHEADING ;

SCR # 70
 0         ( FORM )
 1
 2 CODE XFORM BEGIN, SEC,  BOT LDA, 64 # SBC, BOT STA,
 3                         BOT 1+ LDA, 1 # SBC, BOT 1+ STA,
 4                 0< UNTIL,
 5             BEGIN, CLC, BOT LDA, 64 # ADC, BOT STA,
 6                         BOT 1+ LDA, 1 # ADC, BOT 1+ STA,
 7                 0< NOT UNTIL, RTS, END-CODE
 8
 9 CODE YFORM BEGIN, SEC,  BOT LDA, 200 # SBC, BOT STA,
10                         BOT 1+ LDA, 0 # SBC, BOT 1+ STA,
11                 0< UNTIL,
12             BEGIN, CLC, BOT LDA, 200 # ADC, BOT STA,
13                         BOT 1+ LDA, 0 # ADC, BOT 1+ STA,
14                 0< NOT UNTIL, RTS, END-CODE
15                                                    -->

SCR # 71
 0         ( QUICK-PLOT )
 1   ASSEMBLER HEX
 2   0     VARIABLE X         0 VARIABLE Y           N CONSTANT XL
 3   N 1 + CONSTANT XH    N 2 + CONSTANT SUML    N 3 + CONSTANT SUMH
 4   N 1 - CONSTANT FE
 5 CODE PLOTBODY  ' YFORM JSR, INX, INX, ' XFORM JSR, DEX, DEX,
 6              XSAVE STX, BOT LDY,
 7              BOT 2+ LDA, PHA, BOT 3 + LDA, TAX, PLA,
 8            XL STA, XH STX, TYA, F8 # AND, FE STA, SUML STA,
 9            0 # LDA, SUMH STA, SUML ASL, SUMH ROL, SUML ASL,
10            SUMH ROL, CLC, SUML LDA, FE ADC, SUML STA, SUMH LDA,
11            0 # ADC, SUMH STA, SUML ASL, SUMH ROL, SUML ASL,
12            SUMH ROL, SUML ASL, SUMH ROL, TYA, 7 # AND, CLC,
13            SUML ADC, SUML STA, SUMH LDA, 0 # ADC, SUMH STA,
14            CLC, XL LDA, F8 # AND, SUML ADC, SUML STA, XH LDA,
15            SUMH ADC, SUMH STA, CLC, 0 # LDA, SUML ADC, -->

SCR # 72
 0         ( CONT. )
 1      SUML STA, 20 # LDA, SUMH ADC, SUMH STA, XL LDA, 7 # AND,
 2      7 # EOR, TAX, 1 # LDA,
 3      BEGIN, .A ASL, DEX, 0< UNTIL, .A ROR,
 4      0 # LDY, RTS, END-CODE
 5
 6 CODE PLOTCODE   ' PLOTBODY JSR, SUML )Y ORA,
 7   SUML )Y STA, XSAVE LDX,
 8   INX, INX, INX, INX, RTS, END-CODE
 9 CODE UNPLOTCODE ' PLOTBODY JSR, FF # EOR, SUML )Y AND,
10   SUML )Y STA, XSAVE LDX, INX, INX, INX, INX, RTS, END-CODE
11 CODE ?PLOT  ' PLOTBODY JSR, SUML )Y AND, XSAVE LDX,
12   INX, INX, BOT STA, 0 # LDA, BOT 1+ STA, NEXT JMP, END-CODE
13 CODE ADR ' PLOTBODY JSR, XSAVE LDX, INX, INX,
14          SUML LDA, BOT STA, SUMH LDA, BOT 1+ STA, NEXT JMP,
15          END-CODE    -->

SCR # 73
 0         ( CONT. )
 1
 2 CODE (PLOT) ' PLOTCODE JSR, SEC, SUMH LDA, 20 # SBC, SUMH STA,
 3          SUML LDA, F8 # AND, SUML STA, CLC,
 4 SUMH ROR, SUML ROR, SUMH ROR, SUML ROR, SUMH ROR,
 5 SUML ROR, 4 # LDA, SUMH ADC, SUMH STA, SUML )Y LDA, OF # AND,
 6 SUML )Y STA, COL LDA, .A ASL, .A ASL, .A ASL, .A ASL, SUML )Y
 7 ORA, SUML )Y STA, RTS, END-CODE
 8
 9 CODE  QPLOT  ' PLOTCODE   JSR, NEXT JMP, END-CODE
10 CODE UNPLOT ' UNPLOTCODE JSR, NEXT JMP, END-CODE
11 CODE   PLOT ' (PLOT) JSR, NEXT JMP, END-CODE
12
13 : INIPLOT ;   : -PLOT PLOT ;    : VBREAK ;    : BREAK DROP ;
14
15   DECIMAL

SCR # 74
 0         ( QUICK-LINK )              HEX
 1
 2   0 VARIABLE XO    0 VARIABLE YO    0 VARIABLE X1    0 VARIABLE Y1
 3   0 VARIABLE OF    0 VARIABLE CT    0 VARIABLE DX    0 VARIABLE DY
 4   0 VARIABLE IX    0 VARIABLE IY    0 VARIABLE AX    0 VARIABLE AY
 5
 6
 7 CODE PLOTIT XSAVE LDX,
 8             YO LDA, BOT    STA, YO 1+ LDA, BOT 1+  STA,
 9             XO LDA, BOT 2+ STA, XO 1+ LDA, BOT 3 + STA,
10             ' (PLOT) JSR, RTS, END-CODE
11
12
13
14                                                    -->
15

SCR # 75
 0         ( CONT. )
 1 CODE +STEP SEC, OF     LDA, DX     SBC, OF    STA,
 2                 OF 1+ LDA, DX 1+ SBC, OF 1+ STA,
 3      AY LDA, 0< IF, SEC, XO     LDA, 1 # SBC, XO    STA,
 4                          XO 1+ LDA, 0 # SBC, XO 1+ STA, ELSE,
 5                     CLC, XO     LDA, AY  ADC, XO    STA,
 6                          XO 1+ LDA, 0 # ADC, XO 1+ STA, ENDIF,
 7      IY LDA, 0< IF, SEC, YO     LDA, 1 # SBC, YO    STA,
 8                          YO 1+ LDA, 0 # SBC, YO 1+ STA, ELSE,
 9                     CLC, YO     LDA, IY  ADC, YO    STA,
10                          YO 1+ LDA, 0 # ADC, YO 1+ STA, ENDIF,
11      RTS, END-CODE
12
13 : X<->Y DX @ DY @ DX ! DY !
14        IX @ AY ! IY @ AX !
15        0 DUP IX ! IY ! ;                              -->

SCR # 76
 0         ( CONT. )
 1 CODE LOP
 2 IX LDA, 0< IF, SEC, XO LDA, 1 # SBC, XO STA, XO 1+ LDA,
 3 0 # SBC, XO 1+ STA, ELSE, CLC, XO ADC, XO STA, XO 1+ LDA,
 4 0 # ADC, XO 1+ STA, ENDIF,
 5 AX LDA, 0< IF, SEC, YO LDA, 1 # SBC, YO STA, YO 1+ LDA,
 6 0 # SBC, YO 1+ STA, ELSE, CLC, YO ADC, YO STA, YO 1+ LDA,
 7 0 # ADC, YO 1+ STA, ENDIF,
 8 CLC, OF LDA, DY ADC, OF STA, OF 1+ LDA, DY 1+ ADC, OF 1+ STA,
 9 CT INC, 0= IF, CT 1+ INC, ENDIF,
10
11 OF 1+ LDA, DX 1+ CMP, CS IF, 0= NOT IF, ' +STEP JSR, ELSE,
12 DX LDA, OF CMP, CS NOT IF, ' +STEP JSR, ENDIF, ENDIF, ENDIF,
13 ' PLOTIT JSR, RTS, END-CODE
14                                                    -->
15

SCR # 77
 0         ( CONT. )
 1 CODE LINKCODE DEX, DEX, DEX, DEX, XSAVE STX, ' PLOTIT JSR,
 2              BEGIN, ' LOP JSR,
 3              0 # LDX, CT 1+ LDA, DX 1+ CMP, CS NOT IF, INX,
 4              ELSE, DX LDA, CT CMP, CS IF, INX, ENDIF, ENDIF,
 5              TXA, 0= UNTIL, XSAVE LDX, INX, INX, INX, INX,
 6              NEXT JMP, END-CODE
 7
 8 : LINK Y1 ! X1 ! YO ! XO ! 0 DUP AY ! AX !
 9       X1 @ XO @ 2DUP > IF 1 IX ! ELSE -1 IX ! SWAP ENDIF - DX !
10       Y1 @ YO @ 2DUP > IF 1 IY ! ELSE -1 IY ! SWAP ENDIF - DY !
11       DX @ DY @ > 0= IF X<->Y ENDIF DX @ 2 / OF ! 1 CT !
12       LINKCODE ;
13 : UNLINK ' UNPLOTCODE ' PLOTIT 17 + ! LINK
14          ' (PLOT)     ' PLOTIT 17 + ! ;         DECIMAL
15
```

Listing »Turtle-Grafik« in Forth (Schluß)

x-pert, ein Mini-Experten-System in Forth

x-pert ist ein Mini-Experten-System, das bis zu 2512 Menü- oder Antwort-Blöcke mit 31 Byte Text und bis zu je zehn Menü-Verzweigungen zuläßt. Es bestehen Möglichkeiten zur Erweiterung, Änderung, zum Löschen und zur Mehrfachnutzung von Menüs und Antworten. Der Editor ist auch direkt einsetzbar.

Der Versuch, ein Experten-System auf einem Heimcomputer zu realisieren, scheitert meist an der geringen Speicherkapazität oder an den extrem langen Reaktionszeiten des Diskettenlaufwerks beim Suchen von Datensätzen. Einen vertretbaren Kompromiß gestattet die Programmiersprache Forth, die durch ihr virtuelles Speicherkonzept fast die gesamte Diskette als RAM nutzt. Ein reines Assembler-Programm erlaubt zwar kürzere Verarbeitungszeiten, hält das System aber nicht für Modifikationen und Erweiterungen offen.

In der hier vorgeschlagenen Lösung für den Commodore 64, enthält jeder Datenblock 31 Zeichen Text und bis zu zehn Verzweigungen. Eine Modifikation des Forth-Wortes »r/w« (scr # 8,line # 1) erlaubt es, maximal 162 screens auf eine Diskette zu schreiben. So stehen insgesamt 2512 Datenblöcke zur Verfügung. Um einen direkten Einblick und somit auch direkte Änderungsmöglichkeiten zu schaffen, werden alle Daten im ASCII-Code gespeichert, wobei die Nummern der Folgeblöcke als hexadezimale Zahlen erscheinen. So müssen auch alle Eingaben im hexadezimalen Modus erfolgen.

Neben den üblichen Variationen einen Entscheidungsbaum aufzubauen und zu verändern, erlaubt das vorgestellte System bestehende Menüs oder Antworten in neue Menüs einzufügen. Das führt zwar zu einer Platzersparnis, doch wächst damit die Gefahr der Verfilzung der Äste oder es führt zu Endlosschleifen bei der Darstellung der Baumstruktur.

Im folgenden sollen die neuen, von unserem Mini-System angebotenen Wörter erklärt werden (vergleiche Listing 1).

System-Start

System mit »7 load« von der Programm-Diskette laden. Danach wird eine Daten-Diskette angefordert.

Oder »x-pert« (= = >) eingeben (nach erstmaliger Compilation). Das System wird neu gestartet, ohne eventuell neue oder veränderte Daten aus dem Disk-Buffer zu übernehmen.

Beenden der Arbeit

Nach jeder Sicherung der Daten mit »save« oder »start« kann die Bearbeitung einer Diskette beziehungsweise einer Datei abgeschlossen werden. Das Leuchten der LED am Laufwerk ist ohne Bedeutung. Ein Abschluß mit »done« ist ebenfalls möglich.

Zwischensicherung ohne Protokollveränderung

`save ( ==>)`

Die Datenblöcke im Disketten-Puffer werden auf die Diskette geschrieben und die Bearbeitung vor dem aktuellen Menü fortgesetzt.

`done ( ==>)`

Hat die gleiche Wirkung wie »save«, ruft jedoch kein neues Menü auf.

Zwischensicherung mit Protokollneustart

`start ( ==>)`

Die Datenblöcke werden im Disketten-Puffer auf die Diskette geschrieben und die Bearbeitung mit neuer Protokollierung startet beim aktuellen Menü.

Neue Datendiskette einrichten

`new-disk ( ==>)`

Dieser Vorgang dauert etwa 25 Minuten. Beim Aufruf muß eine eingerichtete Datendiskette oder die Programmdiskette im Laufwerk sein. Bei Aufforderung legt man die neue Diskette ein. Es wird eine Titelzeile angefordert und dann mit dieser Datei gestartet.

Diskette aufräumen

`verify ( ==>)`

Ungenutzte Blöcke werden freigegeben. Dieser Vorgang wird mit zunehmender Anzahl von Daten immer zeitraubender. Man kann ihn auf dem Bildschirm beobachten. Ein fehlerhaftes Ändern mit dem Editor kann »verify« nicht immer korrigieren.

Menü anwählen

`↑(Blocknummer ==>)`

Das Menü beziehungsweise die Antwort mit der eingegebenen Blocknummer erscheint. Die Eingabe kann durch die Positionierung des Cursors unter die entsprechende Menüzeile erfolgen.

Rückwärtsschritt

`←( ==>)`

Es erfolgt die Rückkehr zum vorher aufgerufenen Menü. Die Ebene wird vermindert, im Protokoll sind jedoch alle Schritte festgehalten. »←« dient unter anderem dazu, zum Editor zurückzukehren.

Block zum Menü hinzufügen

Neueintrag

`x+    ( ==>)`

Sofern im Menü und auf der Diskette Platz ist, wird der durch Unterstrich angeforderte Text als neue Menüzeile an der ersten freien Stelle ins Menü eingefügt.

Bestehenden Block einfügen

`x+↑   (Blocknummer ==>)`

Der mit Blocknummer angesprochene Block wird in das aktuelle Menü eingefügt, sofern noch Platz vorhanden ist.

Menü zwischenschieben

`x←  ( Blocknummer ==>)`

Zwischen das aktuelle Menü und die angewählte Menüzeile fügt sich ein Menü ein. Der Text wird durch Unterstreichen angefordert.

Antwort löschen

`x-  ( Blocknummer ==>)`

Der Block wird freigegeben, sofern es sich um einen Antwortblock handelt.

Menü löschen

Ein Menü muß, bevor es gelöscht werden kann, mit

`x=  ( Blocknummer ===>)`

zum Antwortblock umgewandelt werden. Wenn keine Folgeblöcke vorliegen, wird das Menü laut Blocknummer zur Antwort.

Menü in Antwort wandeln

`x=   ( Blocknummer ==>)`

Wenn keine Folgeblöcke vorliegen, wandelt sich das Menü zur Antwort.

Text ändern

`corr   ( Blocknummer ==>)`

Der im angewählten Block gespeicherte Text kann erneut eingegeben werden.

Direktes Ändern mit dem Editor

Die reine ASCII-Darstellung ermöglicht alle Blöcke direkt zu ändern. Natürlich ist hierbei besondere Vorsicht geboten, da man leicht Schleifen oder falsche Verknüpfungen schaffen kann, die später kaum zu entwirren sind.

Hilfe zum Finden eines Blocks

`scr?   ( Blocknummer ==>)`

Es werden die Screen- und Zeilennummer zur hexadezimal eingegebenen Blocknummer dezimal angezeigt und in den Dezimal-Modus umgeschaltet.

Bildschirmprotokoll

`prot   ( ==>)`

Das Protokoll der bisherigen Abfragen wird nach Ebenen gestaffelt auf dem Bildschirm ausgegeben.

Ausdruck des Protokolls

`lprot   ( ==>)`

Geschachtelter Ausdruck der bisher aufgerufenen Menüs und Antworten.

Ausdruck der Gesamtdatei

`print   ( ==>)`

Alle belegten Datenblöcke werden in numerischer Reihenfolge mit Blocknummer, Text, Blockkennung und den Nummern der Folgeblöcke ausgedruckt.

Ausdruck der Baumstruktur

`baum   ( ==>)`

Beginnend ab dem aktuellen Menü als Stamm, wird die Baumstruktur in Form einer Gliederung ausgedruckt. Durch manuelle Menü-Verknüpfungen entstandene Schleifen führen zum Stack-Überlauf und Systemabsturz.

Beschreibung der einzelnen Datenblöcke

Bytes	Inhalt
00-1e	31 Byte Text
1f	Blocktyp-Kennung -: freier Block =: Antwort-Block >: Menü-Block v: Verwaltungs-Block
20-3e	zehn Folgeblock-Nummern jeweils drei ASCII-Zeichen als Hexadezimal-Zahl. Dieses Verfahren ist platzsparend und gestattet eine direkte Editierung.
3f	unbenutzt: blank
20	unbenutzt: Punkt

```
scr # 3
  0 *  MINI-EXPERTEN-SYSTEM: x-pert  *
  1 Verwaltungsblock             >.32062f407
  2 Version: 2.06
  3 (c) Peter Klinghardt Februar 1986
  4 ............................... -12
  5 ............................... -11
  6 ............................... -10
  7 Daten-Diskette einlegen !
  8 Leer-Diskette einlegen !
  9 Beliebige Taste druecken !
 10 noch Folge-Bloecke vorhanden
 11 ist kein Menue-Block
 12 ist kein Antwort-Block
 13 Diskette voll
 14 alle Menue-Zeilen belegt
 15 ............................... - 1
scr # 7
  0 (       Variable und Konstante        )
  1 forth definitions hex
  2 93 emit ." loading: x-pert" cr cr
  3     0 variable men     ( Nummer des aktuellen Menue-Blocks )
  4     0 variable vor
  5     0 variable nach
  6     0 variable addr    ( Zwischenspeicher )
  7     0 variable txt 20 allot ( Text-Bereich )
  8    60 constant anfang ( erster Daten-Block )
  9   a2f constant ende    ( letzter Daten-Block )
 10    60 constant bis     ( letzter benutzter Daten-Block )
 11     0 variable en      ( Menue-Ebene )
 12  a000 constant ev      ( Menue-Eintraege Graphik-Bereich )
 13     0 variable vn      ( Protokoll-Ebene )
 14  b000 constant vv      ( Block-Nummern-Folge Graphik-Bereich)
 15 : m@ men @ ; : m! men ! ; -->
scr # 8
  0 (         Vorabeinstellungen          )
  1 0a2f 1f30 ! ( 162 Screens in r/w )
  2 ." █" ( schwarze Schrift )
  3 : 1f+ 1f + ; : 21+ 21 + ;
  4 code prtof  ( ==>                     )
  5 ( Ausgabe auf Bildschirm              )
  6                xsave stx, ffcc jsr,
  7                xsave ldx, next jmp,
  8 end-code
  9 : prtoff cr prtof
 10 ; graphic &clear
 11 : x0  ( n ==>                         )
 12 ( Block n loeschen                    )
 13 1 ?enough      block dup 40 20 fill
 14              1f+ 2e2d swap ! update
 15 ; -->
scr # 9
  0 -10 message cr
  1 code prton  ( ==>                     )
  2 ( Ausgabe auf Drucker                 )
  3 xsave stx,          4 # lda, 4 # ldx,
  4       7 # ldy, ffba jsr, ( SETLFS )
  5         0 # lda, ffbd jsr, ( SETNAM )
  6                  ffc0 jsr, ( OPEN )
  7       4 # ldx, ffc9 jsr, ( CHKOUT )
  8     xsave ldx, next jmp,
  9 end-code
 10 : cn  ( c ==> b                       )
 11 ( ASCII-HEX in hex-Wert wandeln       )
 12 1 ?enough  dup 20 = if drop 0 else
 13    30 - dup a > if 7 - endif endif
 14 ;
 15 : 0> 0 > ; -->
```

Listing 1. »x-pert«, ein Mini-Expertensystem

Verwaltungsblock-Beschreibung
Blocknummer 31 <hex>

Bytes	Inhalt
00-1e	'Verwaltungsblock'
1f	'v'
20	'.'
21-23	erster Datenblock
24-26	letzter Datenblock
27-29	letzter belegter Block
2a...	Rest ungenutzt

x-pert ist in Forth 64 für den Commodore 64 geschrieben, sollte aber mit nur geringfügigen Modifikationen auf jedem anderen Computer unter FIG-Forth laufen.

Speziell für die Besitzer von Forth 64 sind in Listing 2 die Screens 40 bis 43 gedacht. Sie enthalten eine verbesserte Version der in Forth 64 fehlerhaften Wörter »triad« und eine verbesserte Version des Worts »dump«.

Diese Abänderung belegt keinen weiteren Speicherplatz, da sie in das bestehende System hineingeschrieben wird.

Allerdings ist sie daher auch wirklich nur mit Forth 64 zu verwenden, was jedoch nicht weiter schlimm ist, da diese Fehler ja ebenfalls spezifisch sind.

(Peter Klinghardt/ev)



```
scr # 10
  0 : nc  ( n ==> c                    )
  1 ( hex in ein Byte ASCI umwandeln  )
  2 1 ?enough       dup 9 > if 7 + endif
  3                                 30 +
  4 ;
  5 : x@ ( n ==> addr                  )
  6 ( Adresse des n. Folgeblocks aus  )
  7 ( dem aktuellen Menue              )
  8 1 ?enough       1 - 3 * m@ block
  9        21+ + dup c@ cn 8 <shift swap
 10            1+ dup c@ cn 4 <shift swap
 11                      1+ c@ cn or or
 12 ;
 13 : t?  ( blockaddr ==> ctyp        )
 14 1 ?enough                   1f+ c@
 15 ; -->
scr # 11
  0 : x!  ( b n ==>                    )
  1 ( Block-Nummer b in ASCCI als n.  )
  2 ( Parameter ins Menue eintragen   )
  3 2 ?enough
  4      1 - 3 * m@ block 21+ + addr !
  5 dup f00 and 8 >shift nc addr @  c!
  6  dup 0f0 and 4 >shift nc addr @ 1+
  7  c! 00f and nc addr @ 2+ c! update
  8 ;
  9 : x-.  ( blockaddr ==>             )
 10 1 ?enough          1f -trailing type
 11 ;
 12 : xt?  ( b ==> addr typ           )
 13 1 ?enough          block 1f+ dup c@
 14 ; -->
 15
scr # 12
  0 : header  ( n ==>                   )
  1 ( Block n wird Ueberschrift        )
  2 1 ?enough hex nok 93 emit dup 4 .r
  3  ."   ↑ (  " en @ 4 .r ." . Ebene "
  4       vn @ 5 .r ." . Aufruf )" cr
  5          22 emit block dup t? case
  6    3e of ." Menue: " endof
  7    3d of ." Antw.: " endof
  8    2d of ." frei:  " endof endcase
  9                          1f type cr
 10 ;
 11 : scr?  ( b ==>                    )
 12 ( Block-Nummer als Screen-Zeile   )
 13 1 ?enough decimal dup 4 >shift cr
 14        . ." edit" cr f and . cr
 15 ; -->
scr # 13
  0 : verify  ( ==>                    )
  1      graphic &hi-res vv fff 0 fill
  2          ev fff 0 fill bis anfang do
  3                      ff i ev + !
  4            i m! i block t? 3e = if
  5                   i m! 0b 1 do
  6             ff i x@ vv + c! loop
  7                            endif
  8                             loop
  9                  bis anfang 1+ do
 10             vv i + dup c@ swap
 11                0 swap c! 0= if
 12                     i x0 endif
 13                             loop
 14               &lo-res forth done
 15 ; -->
scr # 14
  0 : x.  ( n ==>                      )
  1 ( Menue-Zeile n ausgeben           )
  2 1 ?enough           22 emit space
  3     dup block dup 1f type t? case
  4    3e of ." Menue" endof
  5    3d of ." Antw." endof
  6    2d of ." frei " endof endcase
  7        cr 21 spaces 4 .r ."  ↑" cr
  8 ;
  9 : prot  ( ==>                      )
 10 ( Protokoll                        )
 11                     cr vn @ 1+ 1 do
 12       vv i 4 * + dup @ swap 2+ @
 13         dup block swap 4 .r ."   "
 14           swap spaces  x-. cr loop
 15 ; -->
scr # 15
  0 : ↑  ( n ==>                       )
  1 ( Menue aus Block n ausgeben.      )
  2 1 ?enough    dup dup dup header m!
  3  en @ 1+ 7ff and dup en ! 2 * ev +
  4  ! vn @ 1+ 3ff and dup vn ! 4 * vv
  5      + dup en @ swap ! 2+ ! 0b 1 do
  6                    i x@ dup 0> if
  7          dup block t? 2d = if
  8              0 i x! drop else
  9                        x. endif
 10                             else
 11                     drop endif
 12                              loop
 13 ;
 14 -e message cr -d message cr cr cr
 15 -->
scr # 16
  0 : lprot  ( ==>                     )
  1 ( Protokoll drucken                )
  2                     cr vn @ 1+ 1 do
  3      vv i 4 * + dup @ swap 2+ @
  4  dup block prton swap 4 .r ."   "
  5      swap 2 * spaces x-. prtoff
  6     ?terminal if leave endif loop
  7 ;
  8 : ←  ( ==>                         )
  9 ( Ein Menue zurueckgehen           )
 10                      en @ 1 > if
 11           en @ 2 - dup en ! else
 12                   0 en ! 0 endif
 13               1+ 2 * ev + @ ↑
 14 ; -->
 15
scr # 17
  0 : x0?  ( ==>                       )
  1 ( Freie Menue-Zeile suchen         )
  2                         0 0b 1 do
  3   i x@ 0= if drop i leave endif
  4                              loop
  5    -dup 0= if -2 message cr endif
  6 ;
  7 : start  ( ==>                     )
  8               m@ dup vv ! dup ev !
  9               done 0 vn ! 0 en ! ↑
 10 ;
 11 : set  ( n ==> n                   )
 12 ( Verwaltung-Block aktualsieren    )
 13 1 ?enough        dup bis max ' bis !
 14               m@ bis 31 m! 3 x! m!
 15 ; -->
scr # 18
  0 : frei?  ( n ==> b                 )
  1 ( sucht einen freien Block ab      )
  2 ( Block n  b=0: kein Block         )
  3 1 ?enough       0 swap ende swap do
  4               i block t? 2d = if
  5               drop i leave endif
  6                              loop
  7 ;
  8 : x?  ( ==> n                      )
  9 ( sucht freien Block               )
 10                bis frei? dup 0= if
 11     drop anfang frei? dup 0= if
 12          cr -3 message cr endif
 13                            endif
 14   -->
 15
scr # 19
  0 : txt?  ( ==>                      )
  1 ( Text einlesen                    )
  2               cr cr txt 1f 20 fill
  3 ." ______________________________"
  4           cr 91 dup emit emit txt 20
  5     expect txt cr dup 20 + swap do
  6                       i c@ 0= if
  7                    20 i c! endif
  8                              loop
  9 ;
 10 : x-pert  ( ==>                    )
 11 cr empty-buffers -10 message 31 m!
 12      1 x@ ' anfang ! 2 x@ ' ende !
 13       3 x@ ' bis ! anfang m! start
 14 ; -->
 15
scr # 20
  0 : replace  ( n ==>                 )
  1 ( Block in Menuezeile n ersetzen   )
  2 1 ?enough        3e m@ block 1f+ c!
  3                      x? dup 0> if
  4     set dup rot x! txt? block dup
  5      txt swap 1f cmove 1f+ dup 3d
  6 swap c! 2+ 1e 20 fill update endif
```

```
  7 ;
  8 : x+  ( ==>                              )
  9 ( Block an Menue anhaengen.              )
 10      x0? -dup 0> if replace endif
 11 ;
 12 : save  ( ==>                            )
 13 ( Aenderungen sichern                    )
 14                                   done ←
 15 ; -->

scr # 21
  0 : new-disk  ( ==>                        )
  1 ( Neue Daten-Diskette einrichten         )
  2          2 list 3 list 4 list 5 list
  3            31 m! 60 dup 1 x! dup 3 x!
  4                dup ' anfang ! ' bis !
  5   a2f dup 2 x! ' ende ! 93 emit -8
  6  message cr -7 message cr key drop
  7                  " n0:x-pert,xp" dos
  8  create-screen 2 2 scopy 3 3 scopy
  9                  20 10 do i x0 loop
 10                             a3  6 do
 11                       1 i scopy loop
 12       ." Titel eingeben " cr 31 m!
 13               1 replace done x-pert
 14 ; -->
 15

scr # 22
  0 : print  ( ==>                           )
  1 ( Ausdruck aller Eintraege               )
  2                  hex bis 1+ anfang do
  3             i m! i block t? 2d = if
  4          i 4 .r ."   frei" cr else
  5  prton i 4 .r space i block 1f
  6     type space i block t? emit
  7   space 0b 1 do i x@ 4 .r loop
  8                       prtoff endif
  9            ?terminal if leave endif
 10                                 loop
 11 ; -->
 12
 13
 14
 15

scr # 23
  0 : tree ( nb ==> nb'                      )
  1   r> drop depth 0= ?terminal or if
  2                           quit endif
  3      dup f000 and c >shift 0b = if
  4                           drop else
  5      dup f000 and c >shift 0= if
  6        1000 + dup 0fff and dup
  7    block prton swap 4 .r depth
  8       2 * spaces x-. prtoff else
  9     1000 + dup 0fff and m! dup
 10   f000 and c >shift 1 - x@ dup
 11                0= if drop endif
 12                            endif
 13                                endif
 14         [ smudge ] tree [ smudge ]
 15 ; -->

scr # 24
  0 : baum  ( ==>                            )
  1 ( Baum des aktuellen Menues              )
  2      prton prtoff clear m@ tree cr
  3 ;
  4 : x-  ( n ==>                            )
  5 ( Antwort-Block freigeben                )
  6 1 ?enough             dup xt? 3d = if
  7                          drop x0 else
  8      cr -4 message cr 2drop endif
  9 ;
 10 : corr  ( n ==>                          )
 11 ( Text im Block veraendern               )
 12 1 ?enough               txt? block txt
 13               swap 1f cmove update
 14 ; -->
 15

scr # 25
  0 : x=  ( n ==>                            )
  1 ( Menue-Block in Antwort wandeln         )
  2 1 ?enough          dup m! xt? 3e = if
  3   0 0b 1 do i x@ + loop 0= if
  4         update 3d swap c! else
  5   cr -6 message cr drop endif else
  6         cr -5 message cr drop endif
  7 ;
  8 : x←  ( n ==>                            )
  9 ( Menue-Blk vor Zeile einfuegen          )
 10 1 ?enough  nach ! m@ vor ! 0b 1 do
 11                    i x@ nach @ = if
 12 i replace 3e i x@ dup m! block
 13   1f+ c! nach @ 1 x! leave endif
 14                       loop vor @ m!
 15 ; -->

scr # 26
  0 : x+↑  ( n ==>                           )
  1 ( bestehendes Menue n anfuegen           )
  2 1 ?enough                   0 0b 1 do
  3                          i x@ 0= if
  4              drop i leave endif
  5                                  loop
  6                           dup 0= if
  7              -2 message 2drop else
  8     x! 3e men @ block 1f+ c! endif
  9 ;
 10
 11  39 block drop
 12  -9 message cr cr
 13  -7 message ."   " cr
 14 close-screen key drop x-pert
 15
```

Listing 1. »x-pert«, ein Mini-Expertensystem (Schluß)

```
scr # 40
  0 (        Korrektur von triad          )
  1 ( Danach arbeitet das Wort wie        )
  2 ( im Handbuch beschrieben             )
  3 ' nop           ( PFA von 'nop'       )
  4   cfa           ( in CFA wandeln      )
  5   dup           ( wird 2x benoetigt   )
  6 ' triad !       ( 1.                  )
  7 ' triad 2+ !    ( und 2. Wort von     )
  8                 ( 'TRIAD' mit 'NOP'   )
  9                 ( ueberschreiben      )
 10
 11
 12 a2f 1f30 !  ( 162 Screens ermoeg-     )
 13             ( lichen, Aenderung       )
 14             ( in r/w                  )
 15 -->

scr # 41
  0 (   Konstante 'dum' einrichten        )
  1
  2 : £ ( addr wert ==> addr+2 ) swap dup 2+ rot rot ! ;
  3 : ££
  4                     2c10 dup dup  ( Addr in dump )
  5             ' dump lfa @ = 0= if  ( dum nicht eingerichtet )
  6                  4483 £ cd55 £    ( Namensfedld eintragen )
  7                    dump lfa @ £   ( lf von dump uebernehmen )
  8                      ' 0 cfa @ £  ( Codefeld fuer Konstante )
  9                             10 £  ( Wert eintragen )
 10          drop ' dump lfa swap £   ( neues Linkfeld fuer dump )
 11                            endif
 12 ;
 13 ££ drop forget £  ( Hilfsworte loeschen )
 14 -->
 15
```

Listing 2. Zwei Verbesserungen zu Forth 64

```
scr # 42
  0 ( dump  von bis ==>  <neue Vers> )
  1 ( dump in der Breite dum mit '.' )
  2 ( als nicht druckbares Zeichen   )
  3 : £ 2 ?enough         1+ base @ hex
  4           -rot swap dup rot swap do
  5       nop cr i 0 5 d.r dum 0 do
  6              dup c@ dup 3 .r
  7          dup 7f and 20 < if
  8                drop 2e endif
  9           pad i + c! 1 + loop   ( Text aufbauen )
 10            space pad dum type   ( Zeilentext ausgeben )
 11 ?terminal if leave endif dum +loop
 12                    drop cr base !
 13 ; -->
 14
 15

scr # 43
  0 (  alte Ver. dump ueberschreiben )
  1
  2 ' £ cfa dup ( Parameter-Feld     )
  3 here swap - ( Laenge             )
  4 ' dump cfa  ( Zieladresse = pfa  )
  5             ( der alten Version  )
  6 swap cmove  ( eintragen          )
  7
  8 forget £    ( Zwischenversion    )
  9             ( loeschen           )
 10 -->
 11 ( !!!! nun geht dump wieder !!!! )
 12
 13
 14
 15
```

Listing 2. Zwei Verbesserungen zu Forth 64 (Schluß)

Am Anfang war das Wort..

Programmieren in Forth bedeutet, neue Wörter zu schaffen, indem bereits definierte Befehle zu immer komplexeren Gruppen zusammengefaßt werden.

Forth ist eine sehr leistungsfähige Sprache. Wer in Forth programmiert, der programmiert nicht; der schafft sich neue Wörter, wobei am Ende oft ein einziges Wort steht, das dann das fertige Programm repräsentiert.

Diese Art des Programmierens bietet gegenüber der herkömmlichen Unterprogramm-Technik erhebliche Vorteile:

- Jedes Wort ist quasi eine Spracherweiterung, das heißt das Computersystem wächst mit dem Programmierer. Je mehr man in Forth programmiert hat, desto häufiger kann man auf Befehle zurückgreifen, die man Monate zuvor geschaffen hat. Dadurch nimmt die Produktivität des Programmierers enorm zu.
- Es ist einfacher, einzelne Wörter auf ihre Richtigkeit zu überprüfen, als ein ganzes Programm. Forth-Programme sind also zuverlässiger und man verliert weniger Zeit bei der Fehlersuche (Debugging).
- Die Programmpflege gestaltet sich erheblich einfacher als in allen anderen Sprachen, da, um ein Programm neu anzupassen, meist nur ganz wenige grundlegende Wörter abzuändern sind. Das ist für all diejenigen wichtig, die zum Beispiel vorhaben, irgendwann auf einen anderen Computer umzusteigen, und Ihre Programmsammlung dann weiter verwenden wollen. Wer einmal versucht hat, Basic-Programme vom C 64 auf einen anderen Computer umzuschreiben, der weiß das zu würdigen.

Daß Forth schwer zu erlernen sei, ist nur ein Gerücht. Forth ist sehr ungewöhnlich, aber auch nicht schwieriger als Basic oder Pascal. Nur wer sich nicht von gewohnten Konzepten trennen kann, mag anfangs Schwierigkeiten haben.

Wie schafft sich der Forth-Programmierer nun seine neuen Wörter? Es gibt prinzipiell drei Möglichkeiten, die die Stichwörter Colon-Definition, Primitive und Compiler umreißen.

Die Colon-Definition

Die Colon-Definition ist die übliche Methode, Wörter zu bauen. Die meisten Programme bestehen aus kaum mehr als einer Reihe von solchen Definitionen.

Eine Colon-Definition besteht aus:
Anfang, Name, Definition, Ende.

Den Anfang kennzeichnet in Forth ein Doppelpunkt (engl. Colon, daher der Name dieser Definitionsart), darauf folgt der Name des zu definierenden Wortes. Die dann folgende Definition ist nichts weiter als eine Liste bereits bekannter Forth-Wörter, die vom System immer dann ausgeführt werden sollen, wenn der neue Name als Anweisung auftaucht. Das Ende der Colon-Definition aktiviert ein Semikolon.

Ein Beispiel:

```
: PETRA 5 + . ;
```

VLIST zeigt, daß PETRA nun ganz oben im Dictionary steht! PETRA addiert eine 5 und zeigt das Ergebnis an. Zum Beispiel ergibt

```
10 PETRA
```

nach Drücken der Return-Taste die Meldung

```
15 OK
```

Die eben durchgeführte Colon-Definition von PETRA repräsentiert natürlich kein besonders gutes Beispiel für sinnvolle Programmierung in Forth, denn normalerweise wird man als Namen für ein Wort immer eine aussagekünftige Bezeichnung wählen (in diesem Falle vielleicht »PLUSFUENF.«). Die Bezeichnung PETRA sollte nur zeigen, daß man in Forth bei der Namensgebung völlig freie Hand hat.

Forth und Maschinensprache

Wer etwas Ahnung von Maschinensprache hat, der sollte nicht denken, »da brauche ich Forth ja nicht mehr«, sondern kann seine Kenntnisse sehr sinnvoll einbringen. Es besteht die Möglichkeit, Wörter statt wie eben in »High-Level«, also als Colon-Definition, auch als sogenannte »Primitive«, das heißt in Maschinensprache, zu schreiben. Dazu benötigt man einen Forth-Assembler, den die meisten Forth-Versionen gleich mitanbieten.

Ein Maschinen-Befehl besteht in der Regel aus Befehlswort und Operanden.

Im 6502-Assembler lautet beispielsweise der Befehl zum Laden des Akkumulativs mit der Konstanten 1

```
LDA #1
```

Der Forth-Assembler benötigt hier die umgekehrte Notation, also erst den Operanden:

```
01 # LDA,
```

(man beachte das Komma hinter LDA!).

Als Beispiel ein kleiner Vergleich:

6502-Assembler

```
$C000 CLC              $C009 RTS
$C001 LDA $D020        $C00A NOP
$C004 ADC  #$01        $C00B NOP
$C006 STA $D020
```

Forth

```
SCR  #  1
 0          (PRIMITIVES)
 1
 2 CODE DANI         CLC,
 3          53280   LDA,
 4              1 # ADC,
 5          53280   STA,
 6           NEXT   JMP,  END-CODE
```

NEXT ist die Einsprungstelle für den »Inneren Interpreter« von Forth. Der Innere Interpreter kümmert sich darum, welches Wort als nächstes an die Reihe kommt, dem »Äußeren Interpreter« fällt die Aufgabe Benutzer und Eingaben zu.

Es ist empfehlenswert, Programme zunächst in High-Level zu erzeugen; falls die Geschwindigkeit dann nicht ausreicht, genügt es meistens, ein oder zwei Worte als Primitive umzuschreiben. Das ist erheblich effektiver, als würde man das ganze Programm in Assembler schreiben. Oft erübrigt sich auch das, da Forth sowieso 100- bis 400-mal schneller als Basic läuft.

Und nun für ganz Raffinierte: Wir können Wörter erfinden, die uns die »Arbeit« des Wörterbauens abnehmen. Dieser Punkt macht Forth so leistungsfähig. Als typischer Vertreter dieser Wörter steht CONSTANT. »n CONSTANT name« schafft ein neues Wort namens »name«, das bei Aufruf n auf den Stack legt. Beispiel:

```
1024 CONSTANT SCREEN
```

definiert zunächst das Wort SCREEN. Der Leser kann sich davon mit Hilfe von VLIST überzeugen.

SCREEN legt dann den Wert 1024 auf den Stapel. Beispiel:

```
SCREEN 500 + .
 1524 OK
```

Anstelle von 1024 (Adresse des ersten Byte des Bildschirmspeichers) kann also überall im Programm SCREEN einspringen.

Will man später das Programm auf einem Computer laufen lassen, bei dem der Bildschirmspeicher vielleicht bei 4096 beginnt, so braucht man nur ein einziges Wort abändern:

```
: SCREEN 4096 ;
```

Praktisch, nicht wahr?

Das Wort CONSTANT schafft also eine ganz neue Klasse von Wörtern, eben mit der Eigenschaft, Konstanten zu sein.

Ein Wort wie CONSTANT muß also zweierlei tun: Erstens das Wort »name« ins Dictionary eintragen, und zweitens »name« sagen, was es tun soll, wenn der Benutzer »name« aufruft.

Zunächst erfinden wir also mit Hilfe einer Colon-Definition ein Wort wie CONSTANT. Nennen wir es ANGEBER. ANGEBER schafft nun eine neue Klasse von Wörtern, nämlich Angeber-Wörter. Das sieht so aus:

```
: ANGEBER <BUILDS DOES> ;
```

Zwischen <BUILDS und DOES> steht die Angabe, wie der Wortkörper auszusehen hat, hier steht also gar nichts; zwischen DOES> und »;« steht, was die Angeber-Wörter dann ausführen sollen, hier also ebenfalls nichts. DOES> bewirkt, daß die PFA (Parameterfeldadresse) des Wortes »name« auf den Stack gelegt wird. Die PFA ist die Adresse des ersten Bytes des Parameterfeldes. Das Parameterfeld beinhaltet eine Liste im Wortkörper, die das Wort zur Ausführung benötigt. Bei Primitives steht da der ganze Maschinencode, bei High-Level-Wörtern besteht das Parameterfeld aus einer Liste der Adressen der Wörter, die abgearbeitet werden sollen, die also in der Colon-Definition standen.

Probieren wir ANGEBER einmal aus:

```
ANGEBER MICHAEL
```

Wir überzeugen uns mit VLIST, daß Michael als neues Wort im Dictionary steht! Nun rufen wir MICHAEL auf. Was passiert? Nichts, außer daß die PFA von MICHAEL auf dem Stack liegt. Deshalb ist MICHAEL ein Angeber: er gibt seine PFA an, aber nichts dahinter!

Probieren wir etwas anderes:

```
: NICHTSNUTZ <BUILDS DOES> DROP ;
```

Wir vermuten, die Nichtsnutz-Wörter entfernen sogar noch ihre PFA. Wir geben zunächst ein:

```
NICHTSNUTZ WALTER
```

und dann

```
WALTER
```

und tatsächlich, nichts passiert.

Versuchen wir einmal, Konstanten bauen zu lassen:

```
: CONSTANTINOPEL <BUILDS , DOES> @ ;
```

Das »,« bewirkt, daß die oberste Zahl vom Stack genommen und ins Dictionary, also in das Parameterfeld, gepackt wird. Die durch CONSTANTINOPEL definierten Wörter holen diese Zahl wegen @dann wieder auf den Stack. Der Vorgang wiederholt sich dauernd. Beispiel:

```
1024 CONSTANTINOPEL SCREEN
SCREEN 500 + .
1524 OK
```

Aha! Prima, aber wir hätten es natürlich auch einfacher haben können:

```
: CONSTANTINOPEL CONSTANT ;
```

Zum Abschluß wollen wir noch ein »richtiges« Programm schreiben, nämlich einen Maskengenerator, der die Bildschirmseite irgendwo speichert und bei Bedarf immer wieder holen kann. Am besten bringen wir den ganzen Bildschirm in einer einzigen Variablen unter. Der Leser wird schon erraten haben: Wir benötigen einen neuen Variablentyp. Nennen wir ihn MASKE. Die Masken sollen in ihrem Parameterfeld zunächst 1000 Leerzeichen, also 1000mal den Bildschirmcode 32 haben. Das erledigt eine Schleife, also in Basic:

```
FOR I=0 TO 999 ......NEXT
```

Das sieht in Forth etwas anders aus:

```
1000 0 DO ......LOOP
```

Die Punkte stehen für eine Anweisung. Nun brauchen wir noch ein Wort, um die 1000 Byte zu transportieren. Dazu bietet uns Forth

```
CMOVE ( a  a  n - )
```

an. CMOVE verlagert n-Byte von der Adresse a nach a.

PFA 1024 1000 CMOVE bringt uns also die Bytes aus der Maske (ab a PFA) in den Bildschirmspeicher.

Den Ausdruck

```
1024 1000 CMOVE
```

kürzen wir durch »M@« ab.

```
MASKE ANDREA
```

kreiert uns eine Leermaske namens ANDREA.

```
ANDREA M@
```

holt uns ANDREA in den Bildschirmspeicher. Zum Speichern mit »M!« werden nur die Adressen a und a vertauscht (siehe Listing). Damit besitzen wir bereits einen primitiven Maskengenerator! Wir können ihn als »Notizzettelspeicher« verwenden, oder als »virtual screen buffer«, oder um mit mehreren Bildschirmen zu arbeiten, und so weiter.

MASKMAKER ist eine etwas komfortablere Version: PAGE löscht den Bildschirm (ASCII-Code 147 beim C 64), KEY erwartet solange eine Tastatureingabe, die mit EMIT auf den Bildschirm gebracht, bis RETURN gedrückt wird. Dann wird die fertige Maske mit »M!« gespeichert.

Wer will jetzt noch behaupten, Forth sei unübersichtlich? Man beachte, daß der ganze Maskengenerator lediglich 127 Byte lang ist, also gerade zwei oder drei Basic-Zeilen entspricht.

(Andreas Carl/ev)

Computer-Ferien: Urlaub einmal anders!

Ein anschauliches Reisemagazin in der Happy-Computer-Ausgabe Juni informiert über Computer-Kurse im Urlaub — *Für jeden Geschmack und jeden Geldbeutel.*

Brandaktuell und bisher nur in Happy-Computer:
Die deutsche Version des spannenden Abenteuerspiels »Im Herzen Afrikas« wird vorgestellt. — Gleich mit dabei:
***Ein Wettbewerb,* in dem Sie als ersten Preis eine Reise nach Kenia gewinnen können.**

***Brandneu:* Das Grafik-Programm »Draughtsman« für *alle CPCs* wird vorgestellt.**
Diesmal im Commodore-Sonderteil:
Ein Kurs zur Spieleprogrammierung in Maschinensprache.
Und »Tron-Construction-Set«, das *Spiel des Monats.*
Eine *Top-Ten-Liste* von *Matrix-Druckern* hilft richtig auszuwählen.
Hardware-Bastelei:* Ausführliche Anleitung zu einem *Sensor-Joystick.
***Im Test:* Die neuen *MSX-Modelle* von Sony.**





erhalten Sie Mitte jedes Monats bei Ihrem Zeitschriftenhändler. Die Juni-Ausgabe erscheint am 12. Mai 1986.

Gutschein

FÜR EIN KOSTENLOSES PROBEEXEMPLAR VON HAPPY-COMPUTER

JA, ich möchte »Happy-Computer« kennenlernen.
Senden Sie mir bitte die aktuellste Ausgabe kostenlos als Probeexemplar. Wenn mir »Happy-Computer« gefällt und ich es regelmäßig weiterbeziehen möchte, brauche ich nichts zu tun: Ich erhalte »Happy-Computer« dann regelmäßig frei Haus per Post und bezahle pro Jahr nur DM 66,— statt DM 72,— Einzelverkaufspreis (Ausland auf Anfrage).

Vorname, Name

Straße | PLZ, Ort

Datum | 1. Unterschrift

Mir ist bekannt, daß ich diese Bestellung innerhalb von 8 Tagen bei der Bestelladresse widerrufen kann und bestätige dies durch meine zweite Unterschrift. Zur Wahrung der Frist genügt die rechtzeitige Absendung des Widerrufs.

Datum | 2. Unterschrift

Gutschein ausfüllen, ausschneiden, in ein Kuvert stecken und absenden an:
Markt&Technik Verlag Aktiengesellschaft, Vertrieb, Postfach 1304, 8013 Haar

HCS586

Pilot für Höhenflüge

Hier finden Sie eine Einführung in die Programmiersprache Pilot. Anhand einiger Beispiele werden die Fähigkeiten dieser interessanten Sprache vorgestellt.

Pilot hat nichts mit dem gleichlautenden Wort »Pilot« zu tun, sondern ist die Abkürzung für »Programmed Inquiry, Learning Or Teaching«, auf deutsch etwa »Programmiertes Abfragen für Lernen und Lehren«. Der Zweck ist also klar: Pilot wurde entwickelt, um schnell und einfach Lernprogramme zu schreiben. Lernprogramme, die mit Computern nichts zu tun haben müssen! Nein, alles ist denkbar, von Vokabelabfragen über Rechtschreibübungen bis hin zu Geschichts-, Biologie- oder Physik-Lernprogrammen. Eine solche Vielfalt muß nicht unbedingt erstaunen, denn alle derartigen Programme benutzen ja ein bestimmtes Konzept immer wieder, nämlich das der Stringverarbeitung. Und gerade auf diesem Gebiet liegt die Stärke von Pilot. Die Sprache hat einige Befehle zu bieten, die nicht alltäglich sind.

Doch gehen wir der Reihe nach vor und beginnen wir mit den eher gewöhnlichen Befehlen.

Hallo Pilot, wie geht's?

Jede Anweisung in Pilot hat die folgende Form:

Befehl: Operandenliste

In jeder Zeile steht genau eine Anweisung, Zeilennummern gibt es nicht. Ein Beispiel: Das Ausgeben von Text auf dem Bildschirm geschieht durch den Befehl »T« (für »Type«). Also schreibt:

```
T:Hallo Pilot
```

die Worte »Hallo Pilot« auf den Bildschirm.

Will man den Type-Befehl in mehreren Anweisungen hintereinander benutzen, muß man den Befehl nicht mehrfach angeben. Dann genügen die Doppelpunkte.

```
T:Hallo Pilot
:Wie gehts
```

schreibt erst »Hallo Pilot« und dann »Wie gehts« eine Zeile darunter.

Der Type-Befehl existiert in zwei Varianten.

»TS« löscht zuerst den Bildschirm und gibt dann den Text aus und »TH« (»Type and Hang«) läßt den Cursor für den nächsten Type-Befehl in derselben Zeile stehen.

Ähnlich wird der A-Befehl (für »Accept«) verwendet. »A:#X« entspricht der Basic-Anweisung INPUT X. Das Zeichen »#« gibt an, daß X eine numerische Variable ist.

Als Variante ist »AS« (»Accept Single Character«) zu erwähnen. Dabei wartet der Computer nur auf die Eingabe eines einzelnen Zeichens und fährt dann sofort mit dem Programmlauf fort.

Die Befehle Type und Accept dienen also zur Kommunikation zwischen Programm und Benutzer. Aber auch Berechnungen fallen an. Dazu dient der Befehl »C« (für »Compute«).

»C:x=3*2« berechnet beispielsweise das Produkt aus 3 und 2 und weist es der Variablen x zu. Bei dem Compute-Befehl benötigt man das »#« nicht.

Zu diesem Thema wäre noch zu sagen, daß es in Pilot nur ganze Zahlen gibt. Wer das als Nachteil empfindet, sollte sich noch einmal klarmachen, für welchen Zweck diese Sprache überhaupt konzipiert wurde, denn ein Rechtschreib- oder Geschichtsprogramm benötigt nun wirklich keine Kommazahlen.

Immerhin sind aber für ganze Zahlen die Operatoren »+«, »–«, »*« und »/« vorhanden.

Schließlich benötigen wir einen Sprung-Befehl. Dieser lautet J (für »jump«). Dazu muß natürlich ein Sprungziel angegeben werden. Basic erledigt diese Aufgabe durch eine Zeilennummer, in Pilot hingegen geschieht es anschaulicher durch Angabe eines »Labels«, das heißt eines symbolischen Namens.

»J:end« springt also zum Label mit dem Namen »end«. Dieses Label muß in dem Programm vorhanden sein, und zwar in der Form »*end«.

Labels statt Zeilennummern

Einen bedingten Sprung bewirkt die Angabe einer Bedingung: »J(x=0):end« löst einen Sprung nach »end« nur unter der Voraussetzung aus, daß X den Wert Null hat. Ansonsten hat diese Anweisung keine Wirkung. In der Klammer zwischen »J« und »:« kann ein beliebiger logischer Ausdruck stehen. An Vergleichsoperatoren fehlt keiner der von Basic her gewohnten.

Übrigens kann man jeden Befehl von solch einer Bedingung abhängig machen: »T(x=0):gleich 0« gibt den Text nur aus, wenn x den Wert 0 hat, »C(a=b):y=x+3« berechnet y=x+3, falls a=b gilt.

Damit besitzen wir genügend Wissen, um ein erstes Programm zu verstehen (Listing 1). Es handelt sich dabei um ein Programm, das entscheidet, ob eine eingegebene Zahl eine Primzahl ist oder nicht. Zum direkten Vergleich steht hinter jeder Zeile des Programms die entsprechende Basic-Zeile.

Das erste Programm

Als Primzahl wird eine Zahl bezeichnet, die nur durch 1 und sich selbst ohne Rest teilbar ist. Wollen wir also prüfen, ob eine Zahl diese Eigenschaft besitzt, untersuchen wir alle in Frage kommenden Zahlen auf die Teilereigenschaft hin. Das sind natürlich nur die Zahlen, die kleiner sind als die zu untersuchende. Nennen wir unsere Zahl einmal x. Dann beginnen wir bei i=2 zu testen, ob i Teiler von x ist. Die Bedingung hierfür lautet: X/i*i=x. Man beachte, daß x/i eine ganzzahlige Division ist, das heißt Kommastellen entfallen. Machen Sie sich anhand von Beispielen klar, daß diese Bedingung gleichwertig zur Teilerbedingung ist. Sie ist bis i=x-1 zu überprüfen, dann nicht mehr.

Sobald sich die Bedingung erfüllt, ist i ein Teiler von x und daher x keine Primzahl. Dann erfolgt ein Sprung nach »*notprim«. Ansonsten läuft die Schleife unterhalb von »*test« bis zum Ende und gelangt dann nach »*prim«. Was in den einzelnen Fällen passiert, sollte jedem klar sein.

Als Sonderfall gilt x=2, da die Schleife unterhalb von »*test« bei i=2 zu laufen beginnt und daher sofort 2 als Teiler von 2 finden würde.

Vergleichen Sie bitte Pilot- und Basic-Programm Zeile für Zeile, um sich die Unterschiede deutlich zu machen.

Eine Anmerkung zum Schluß: Die Anweisung »W:20« stellt lediglich eine Warteschleife dar (W entspricht »Wait«) und hat sonst keinerlei Funktion. Sie ist aber nötig, da der Pilot-Interpreter nach Beendigung eines Programms sofort wieder ins Hauptprogramm zurückspringt. Die Wait-Anweisung gibt dem Anwender des Programms etwas Zeit, das Resultat zu lesen.

Nun kommen wir endlich zu den angekündigten Besonderheiten von Pilot. Diese betreffen zunächst den Accept-Befehl, da er auch ohne Operand aufgerufen werden kann, einfach in der Form »A:«. Die Eingabe geht damit natürlich nicht verloren, sonst hätte das alles wenig Sinn. Vielmehr befindet sie sich in einem speziellen Eingabe-Buffer. Mit »%b« (für Buffer) können sie nun darauf zurückgreifen. Wertzuweisungen oder Berechnungen sind zum Beispiel durch »C:x=%b« möglich oder die Eingabe wird wieder durch »T: #%b« ausgegeben (man beachte das #-Zeichen).

Das scheint natürlich nicht sensationell und ist daher auch nicht der eigentliche Zweck der Angelegenheit. Der folgt jetzt in Form des Befehls M (für »Match«). Die Erfinder von Pilot machten sich nämlich Gedanken, wie sich solche Frage- und Antwortspiele, wie sie Vokabel-Lern-Programme nun einmal darstellen, mit dem geringsten Aufwand programmieren lassen. Es ist ja überhaupt nicht nötig, dem Eingabestring einer Vokabel einen Variablennamen zu geben. Man muß eben nur feststellen können, ob die Eingabe mit der korrekten Lösung übereinstimmt. Dieses Testen übernimmt der Match-Befehl (match heißt ja übereinstimmen).

Die Anweisung »M:super« prüft also, ob die letzte Eingabe (durch eine unspezifizierte Accept-Anweisung) mit dem Wort »super« übereinstimmt. Aber wieder ist nicht klar, was mit dem Ergebnis geschieht, da ja zwei Fälle auftreten können.

Y (für »yes«) heißt: Die Eingabe stimmt überein.

N (für »no«) heißt: Die Eingabe stimmt nicht überein.

Wann immer nun bei einer der folgenden Anweisungen ein Y (beziehungsweise N) hintenansteht, wird diese nur ausgeführt, wenn eine Übereinstimmung festgestellt wurde (beziehungsweise nicht festgestellt wurde).

```
ts:Welches Gebaeude ist auf dem
:1000-Mark-Schein zu sehen ?
a:
m:Limburg
thy:Richtig,
jy:end
t:Falsch.
:Ein Tip : Es handelt sich um den
:Dom einer hessischen Stadt.
a:
m:Limburg
thy:Richtig,
jy:end
th:Leider falsch,
*end
t: es ist der Limburger Dom
w:20
```

Listing 2. Demonstrationsprogramm für den Accept- und Match-Befehl

Zum Beispiel: »TY: Richtig« gibt »Richtig« auf den Bildschirm, wenn die letzte Match-Anweisung ein positives Ergebnis brachte.

Jeder mit Y oder N versehene Befehl, sei es nun TN, JY oder CN, bezieht sich also immer auf das Ergebnis der letzten Match-Anweisung.

Unser neues Wissen wollen wir uns an einem weiteren Pilot-Programm ansehen (Listing 2). Da dieses die speziellen Accept- und Match-Anweisungen benutzt, kann es nicht einfach Zeile für Zeile in ein Basic-Programm übertragen werden. Aber es ist trotzdem nicht schwierig zu verstehen.

Der Benutzer wird gefragt, welches Gebäude ein 1000-Mark-Schein zeigt. Es handelt sich hierbei um den Limburger Dom.

Betrachtet man die Match-Anweisung, die hier Übereinstimmung feststellen soll, so sieht man, daß nur ein Vergleich mit »Limburg« stattfindet. Das hat durchaus seine Vorteile. Denn der Benutzer, der die Antwort kennt, kann nun »Limburger Dom«, »Der Limburger Dom« oder auch nur »Limburg« eingeben.

Alles wird hier als richtig erkannt, da jede Antwort das Vergleichswort »Limburg« enthält. Derartige Flexibilität kostet in anderen Programmiersprachen wesentlich mehr Mühe als hier.

War die Antwort korrekt, wird »Richtig« ausgegeben und zum Ende gesprungen.

Pilot	Basic
`*start`	`120 rem start`
`  ts:Geben Sie eine Zahl ein`	`110 rem`
`  a:#x`	`120 input x`
`  j(x=2):prim`	`130 if x=2 then 190`
`  c:i=2`	`140 i=2`
`*test`	`150 rem test`
`  j(x/i*i=x):notprim`	`160 if int(x/i)*i=x then 220`
`  c:i=i+1`	`170 i=i+1`
`  j(i<x):test`	`180 if i<x then 150`
`*prim`	`190 rem prim`
`  t:#x  ist prim`	`200 print x;" ist prim"`
`  j:end`	`210 goto 240`
`*notprim`	`220 rem notprim`
`  t:#x  ist nicht prim`	`230 print x;" ist nicht prim"`
`*end`	`240 rem end`
`  w:20`	`250 rem pause`

Listing 1. Vergleich: Primzahlenberechnung in Pilot und Basic

Übersicht über die Pilot-Befehle

Befehl	Bedeutung
T:Text	Ausgabe eines Textes auf dem Bildschirm
TS:Text	Löschen des Bildschirms, dann Ausgabe
TH:Text	Ausgabe eines Textes ohne Carriage Return
A: #num.Var.	Eingabe einer numerischen Variablen
AS: #num.VAR.	Eingabe einer einstelligen Zahl
A:	Eingabe in den Eingabe-Buffer
AS:	Eingabe eines Zeichens in den Buffer
C:Var=Ausdr.	Auswerten des numerischen Ausdrucks und Zuweisung des Wertes an die Variable
J:Label	Sprung zu einem Label
*Label	Kennzeichnung eines Labels
M:Text	Vergleich der letzen Eingabe mit dem Text
Befehl(Bedingung):Operanden	Bedingte Anweisung. Die Bedingung muß ein logischer Ausdruck sein
BefehlY:Operanden	Befehl wird ausgeführt, wenn die letzte M-Anweisung positiv ausfiel.
BefehlN:Operanden	Befehl wird ausgeführt, wenn die letze M-Anweisung negativ ausfiel.

Aus Limburg kommt nicht nur Käse

Bei falscher Antwort bekommt der Benutzer einen Tip und darf ein zweites Mal antworten. Danach erscheint es nicht mehr so unnatürlich, nur mit »Limburg« zu antworten, und so ist die Anweisung »M:Limburg« hier durchaus gerechtfertigt.

Gehen Sie das Programm nun noch einmal durch und achten Sie besonders auf die Anwendung von A, M, Y, N.

Kenner von Pilot meinen nun wohl zu dem Programm, daß man es noch kürzer und einfacher hätte schreiben können. Das stimmt natürlich. Pilot kennt noch mehr ungewöhnliche Anweisungen, die zur Vereinfachung beitragen können. Aber die in diesem einführenden Artikel vorgestellten Mittel erlauben es nicht. Es sollte Ihnen hier ja auch nur ein Überblick vermittelt werden, was Pilot leistet und wie es arbeitet. Einen Vergleich von Pilot gegenüber anderen Programmiersprachen sollten Sie jetzt aber zumindest ziehen können und sich dann entscheiden.

(Eckart Winkler/ev)

Tiny Pilot zum Abtippen

Die Programmiersprache Pilot eignet sich besonders für Lehrprogramme. Hier ist ein Programm, das einen ersten Eindruck dieser Sprache vermittelt.

Pilot wurde erfunden, um auf besonders einfache Art und Weise Lehrprogramme zu entwickeln. Auf dieses Ziel abgestimmt ist daher auch die gesamte Struktur der Sprache. Pilot bietet einfache Befehle zur Ausgabe von Fragen, zur Eingabe von Antworten und zum Vergleich der Antworten mit vorgegebenen Texten. Zwei Flags, nämlich »Y« (für Yes) und »N« (für No) speichern den Wahrheitswert der Antworten und beeinflussen in sehr einfacher Weise den Programmablauf.

Der hier vorgestellte Tiny-Pilot-Interpreter lehnt sich an die Struktur der Sprache Pilot an, verfügt jedoch nur über einen eingeschränkten Befehlssatz. Die Befehle sind auch von der Komplexität und Leistungsfähigkeit her zum Teil wesentlich einfacher aufgebaut, als das in größeren Pilot-Versionen der Fall ist. So sind Rechenoperationen nur sehr eingeschränkt möglich, Variable und Unterprogramm-Namen fehlen.

Dieser Pilot-Interpreter soll denn auch kein professionelles Programmierssystem für Pilot darstellen, sondern lediglich einen ersten Eindruck dieser Sprache vermitteln. Vermißt jemand wesentliche Funktionen von Pilot, dann kann er diese ohne große Probleme selbst einbauen – der Interpreter ist schließlich in Basic geschrieben und extra auf Erweiterungsfähigkeit angelegt. Das Programm (Listing 1) besteht intern eigentlich aus zwei Teilen, nämlich dem Editor, mit dem Pilot-Programme geschrieben, gespeichert und verändert werden, und dem eigentlichen Pilot-Interpreter, der Tiny-Pilot-Programme ausführt. Obwohl das Programm für den Schneider CPC 464 geschrieben ist, läßt es sich ohne Schwierigkeiten auch an andere Computersysteme anpassen.

Der Editor

Der hier realisierte Editor orientiert sich am normalen Basic-Editor, das heißt, Programmzeilen werden, beginnend mit einer Zeilennummer, eingegeben. Durch Eingabe nur einer Zeilennummer (ohne weiteren Text) wird die entsprechende Zeile gelöscht. Weitere Kommandos werden im folgenden vorgestellt. Dabei ist zu beachten, daß ein Befehl und ein eventuell angegebener Parameter (Zeilennummer oder Programmname) immer durch genau ein Leerzeichen (Space) getrennt sein müssen. Programmnamen stehen im Gegensatz zu Basic nicht in Anführungszeichen.

LIST nr: Listet Pilot-Programm. Die Angabe einer Zeilennummer (nr) bewirkt die Ausgabe des Listings ab dieser Zeile. Bei fehlender Zeilenangabe beginnt das Listing mit der ersten vorhandenen Zeile. Drücken der Space-Taste hält die Auflistung an, durch Betätigen irgendeiner anderen Taste wird das Listing fortgesetzt. Eine beliebige Taste (außer Space) während des Auflistens unterbricht das Listing total.

LOAD name: Lädt ein Pilot-Programm von Kassette oder Diskette. Der Programmname wird mit der Erweiterung »PLT« für Pilot versehen, falls er nicht schon eine Erweiterung enthält. Beispiel: Der Befehl »LOAD TEST« sucht und lädt eine Datei mit Namen »TEST.PLT«. Pilot-Programme werden als sequentielle Dateien abgelegt. Das Öffnen einer sequentiellen Datei zum Lesen geschieht im Schneider-Basic mit dem Kommando »OPENIN "Name"«, geschlossen wird die Datei mit »CLOSEIN«. Das eigentliche Lesen aus der Datei geht mit »INPUT # 9« beziehungsweise »LINE INPUT # 9« vor sich. Diese Befehle unterscheiden sich von Computer zu Computer. Wenn Sie dieses Programm auf einem anderen Computer als einen Schneider CPC laufen lassen wollen, dann schlagen Sie bitte die entsprechenden Kommandos für Ihren Computer nach.

SAVE name: Dieser Befehl speichert ein Pilot-Programm unter dem angegebenen Namen auf Kassette oder Diskette. Hinsichtlich Programmnamen und Anpassung an andere Computer gilt auch hier sinngemäß das bei LOAD Gesagte. Eine sequentielle Datei wird beim Schneider CPC mit »OPENOUT "Name"« zum Schreiben geöffnet und mit »CLOSEOUT« wieder geschlossen. Das eigentliche Schreiben in die Datei geschieht mit »PRINT # 9«.

EXIT: Dieses Kommando beendet das Arbeiten mit dem Pilot-Interpreter, indem es einfach ins Basic zurückkehrt. Vergessen Sie nicht, Ihr Pilot-Programm vor EXIT zu speichern.

CAT: Dieses Kommando listet das Inhaltsverzeichnis einer Diskette auf dem Bildschirm. Beim Schneider CPC geschieht dies ganz einfach durch das gleichlautende Basic-Kommando CAT, bei anderen Systemen kann das entsprechende Kommando anders lauten (DIR, DIRECTORY, CATALOG oder ähnlich). Einige ältere Computermodelle benötigen an dieser Stelle ein ganzes Unterprogramm.

NEW: Wie in Basic, so löscht dieser Befehl auch hier ganz einfach das im Speicher befindliche Pilot-Programm. Und wie in Basic sollte man diesen Befehl daher nur mit Überlegung anwenden.

RUN: Ebenfalls ganz analog zu Basic veranlaßt RUN den Pilot-Interpreter, das im Speicher stehende Pilot-Programm auszuführen.

Wer möchte, dem steht nichts im Wege, den Editor-Teil selbst um weitere Kommandos zu erweitern, etwa um einen RENUMBER-Befehl oder um ein Kommando für das Listen des Pilot-Programms auf einem Drucker.

Die Tiny-Pilot-Befehle

Nach RUN beginnt der Pilot-Interpreter, der im Listing die Zeilen ab 1000 belegt, mit der Abarbeitung des Programms. Die Variable PC spielt dabei die Rolle eines Programmzählers. Sie enthält nämlich immer die Nummer der gerade aktuellen Befehlszeile. Nach der Ausführung eines Befehls wird PC um eins erhöht und zeigt auf die nächste Programmzeile. Ist diese Zeile leer, dann wird PC solange weiter erhöht, bis eine Befehlszeile gefunden wurde.

Die Ausführung eines Befehls geschieht in drei Schritten. Als erstes wird getestet, ob die Befehlszeile mit den Buchstaben »Y« oder »N« anfängt. Ist dies der Fall, dann wird der Befehl in der Zeile nur dann ausgeführt, wenn das entsprechende Flag (Y oder N) gesetzt ist. Y und N beeinflußt in erster Linie der Vergleichsbefehl. War eine Antwort richtig, dann wird das Y-Flag gesetzt, war die Antwort falsch, das N-Flag.

Nach dem Test, ob der Befehl überhaupt ausgeführt werden soll, erfolgt nun als zweites die Befehls-Analyse. Für jeden Pilot-Befehl steht ein einzelner Buchstabe oder ein Sonderzeichen. Die Befehlszeile wird also zerlegt in ein Befehlszeichen (Variable A$) und einen Operandenstring (Variable B$).

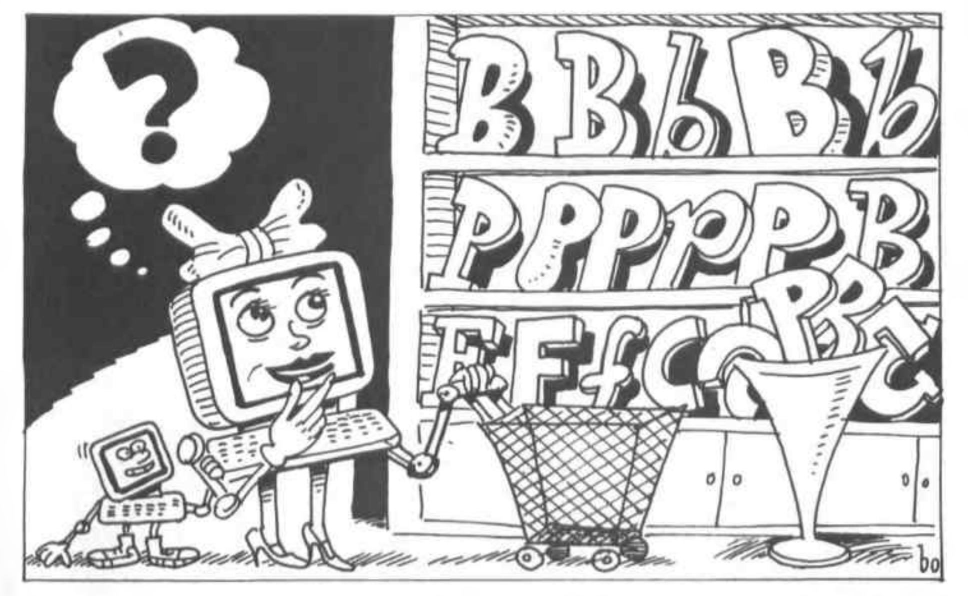
Hinweise zum Anpassen des Programms an andere Computer

DEFINT A-Z	Definiert alle numerischen Variablen als Integer. Kann fortgelassen werden.
CLS	Löscht den Bildschirm und bringt den Cursor in die linke obere Ecke.
MID$(A$,n)	Eine besondere Form der MID$-Funktion, die ja normalerweise mit drei Parametern aufgerufen wird. Ergibt alle Zeichen von A$ an der n-ten Position. Obwohl wenig bekannt, funktioniert diese Form bei praktisch allen Computern, die die MID$-Funktion überhaupt kennen.
PRINT USING	Eine formatierte PRINT-Anweisung. Hinter USING folgt ein Format-String, der die Feldbreite für das Ausdrucken einer folgenden numerischen Variablen enthält. USING und Formatstring können fortgelassen werden, allerdings ergibt sich dann natürlich ein unformatierter Ausdruck.
INSTR(A$,B$)	Eine spezielle String-Funktion. Der String B$ wird innerhalb des Strings A$ gesucht. Die Funktion liefert als Ergebnis die Stelle, an der B$ in A$ auftaucht. Ist B$ in A$ nicht enthalten, dann ist das Ergebnis Null. Beispiel: »INSTR("ABCD","C")« liefert als Ergebnis den Zahlenwert 3, denn »C« taucht in »ABCD« an der dritten Stelle auf. Falls INSTR in einer Basic-Version nicht vorhanden ist, muß die Funktion durch ein Unterprogramm ersetzt werden.

ELSE	Gegenstück zu THEN. Wenn die Bedingung in einer IF-Anweisung nicht erfüllt ist, werden die Anweisungen hinter ELSE ausgeführt. Bei Computern, die ELSE nicht kennen, muß man eine solche IF-Anweisung entsprechend aufteilen.
	Beispiel:
	`IF A=B THEN C=D ELSE E=F`
	hat die gleiche Wirkung wie die beiden Zeilen
	`IF A=B THEN C=D`
	`IF A()B THEN E=F`

INKEY$	Die Funktion wird in der Form »A$=INKEY$« angewendet und ergibt die gerade gedrückte Taste. Es wird nicht auf das Drücken einer Taste gewartet, sondern ein Leerstring zurückgeliefert, wenn gerade keine Taste betätigt ist. Entspricht »GET A$« bei einigen Computern.
LINE INPUT	Wie INPUT, es können aber auch Kommata und so weiter eingegeben werden. Muß für einige Computer durch Unterprogramm realisiert werden.
CAT	Zeigt Disketten-Directory am Bildschirm an. Entspricht DIR, DIRECTORY, FILES, CATALOG und so weiter bei anderen Computern.
OPENOUT "Name"	Öffnet sequentielle Datei zum Schreiben.
PRINT #9	Schreibt in sequentielle Datei.
CLOSEOUT	Schließt Schreib-Datei.
OPENIN "Name"	Öffnet sequentielle Datei zum Lesen.
INPUT #9	Liest aus sequentieller Datei.
LINE INPUT #9	Liest einen ganzen Datensatz komplett mit Kommata und so weiter.
CLOSEIN	Schließt Lese-Datei.

Übersicht der Schneider-spezifischen Befehle und ihrer Wirkung



Es erfolgt eine Verzweigung in verschiedene Routinen je nach Befehl, und endlich als dritte Phase die eigentliche Befehlsausführung. Um dem Interpreter die Arbeit zu erleichtern (und damit die Abarbeitungsgeschwindigkeit zu erhöhen) fordern die Pilot-Befehle ein bestimmtes Eingabeformat. Falls eines der Flags Y oder N angegeben ist, dann muß das Befehlszeichen direkt ohne Zwischenraum auf den Flag-Namen folgen. Hinter dem Befehlszeichen ist genau ein Leerzeichen vorgesehen, es wird aber nicht getestet, um was für ein Zeichen es sich dabei handelt. In der Regel drückt man die Leertaste (Space) nach dem Befehl, es ist aber auch ein beliebiges anderes Zeichen möglich. Insbesondere kann (muß aber nicht) beim T-Befehl zur Textausgabe ein Anführungszeichen verwendet werden. Falls ein Befehl einen Parameter (Zeilennummer, Text oder Zahlenwert) benötigt, muß dieser an der übernächsten Position hinter dem Befehlssysmbol beginnen.

Nach dieser Vorrede nun endlich die Pilot-Befehle im einzelnen:

: text Kommentar – der folgende Text wird vom Interpreter ignoriert.

T text Textausgabe – der nachfolgende Text wird am Bildschirm angezeigt. Wenn das letzte Textzeichen ein Semikolon ist, wird nach der Textausgabe keine neue Zeile begonnen (das Semikolon wird nicht mit ausgedruckt). Eine weitere Textausgabe schließt dann unmittelbar an diesen Text an. Ist das letzte Zeichen kein Semikolon, beginnt nach der Textausgabe eine neue Zeile. Ein »T« allein erzeugt eine Leerzeile.

A Antwort – Pilot wartet auf eine Texteingabe vom Benutzer. Diese Eingabe wird in einem speziellen Antwort-Speicher (im Programm durch die Variable C$ realisiert) zur weiteren Verwendung bereitgehalten. Der Antwortspeicher enthält immer den mit dem letzten A-Kommando eingegebenen Text, alle früheren Antworten gehen durch ein A-Kommando verloren.

V text Vergleich – prüft, ob die zuletzt eingelesene Antwort mit dem angegebenen Text übereinstimmt. Falls ja, dann wird das Y-Flag gesetzt und der »Match-Zähler« (er dient zum Zählen der richtigen Antworten) um eins erhöht. Ist die letzte Antwort nicht mit dem Vergleichstext identisch, dann wird das N-Flag gesetzt und der Match-Zähler bleibt unverändert.

W Wiederhole letzte Antwort – druckt die zuletzt mit A eingelesene Antwort wieder aus.

G Glocke – erzeugt ein kurzes akustisches Signal (entspricht der Basic-Anweisung »PRINT CHR$(7)«), das beispielsweise für Fehlermeldungen genutzt werden kann.

```
1 REM TINY PILOT INTERPRETER
2 REM ----------------------
3 REM
4 REM (c) Volker Everts, 1986
5 REM
10 DEFINT A-Z
20 NN=199:REM max. Anzahl Programmzeilen
30 DIM P$(NN+1):P$(NN+1)="E":REM Pilot-P
rogramm
40 CLS:PRINT TAB(6);"*** TINY PILOT ***"
:PRINT
45 PRINT"OK"
50 PRINT">";:LINE INPUT A$:A$=A$+" "
60 IF LEFT$(A$,1)=" " THEN A$=MID$(A$,2)
:GOTO 60
70 IF A$="" THEN 50
80 I=INSTR(A$," ")
90 B$=MID$(A$,I+1):A$=LEFT$(A$,I-1)
100 IF VAL(A$)>0 OR A$="0" THEN GOSUB 20
0:GOTO 50:REM Zeilennummer
110 IF A$="LIST" THEN GOSUB 300:GOTO 45
120 IF A$="LOAD" THEN GOSUB 400:GOTO 45
130 IF A$="SAVE" THEN GOSUB 500:GOTO 45
140 IF A$="EXIT" THEN GOSUB 600:GOTO 45
150 IF A$="CAT" THEN GOSUB 700:GOTO 45
160 IF A$="NEW" THEN GOSUB 800:GOTO 45
170 IF A$="RUN" THEN GOSUB 1000:IF E THE
N 180 ELSE 45
175 PRINT"+++ UNBEKANNTES KOMMANDO":GOTO
 45
180 PRINT"+++ FEHLER IN PILOT-PROGRAMM:"
:PRINT PC;P$(PC):E=0:GOTO 45
185 PRINT"+++ FALSCHE ZEILENNUMMER":GOTO
 45
197 REM
198 REM Zeile einfuegen/loeschen
199 REM
200 N=VAL(A$):IF N>NN THEN 185
210 P$(N)=B$
220 RETURN
297 REM
298 REM Pilot-Programm listen
299 REM
300 IF B$="" THEN B$="0"
310 FOR I=VAL(B$) TO NN
320 IF P$(I)<>"" THEN PRINT USING"####";
I;:PRINT" ";P$(I)
330 K$=INKEY$:IF K$="" THEN 360
340 IF K$<>" " THEN I=NN:GOTO 360
350 K$=INKEY$:IF K$="" THEN 350
360 NEXT
370 RETURN
397 REM
398 REM Pilot-Programm laden
399 REM
400 I=INSTR(B$,"."):IF I=0 THEN B$=B$+".
PLT"
410 GOSUB 800:REM altes Programm loesche
n
420 OPENIN B$:REM sequentielle Datei oef
fnen
430 INPUT#9,N:IF N=-1 THEN CLOSEIN:RETUR
N
440 LINE INPUT#9,P$(N):GOTO 430
497 REM
498 REM Pilot-Programm speichern
499 REM
500 I=INSTR(B$,"."):IF I=0 THEN B$=B$+".
PLT"
510 OPENOUT B$:REM sequentielles File oe
ffnen
520 FOR I=0 TO NN
530 IF P$(I)<>"" THEN PRINT#9,I:PRINT#9,
P$(I)
540 NEXT
550 PRINT#9,-1:CLOSEOUT
560 RETURN
597 REM
598 REM Programm beenden
599 REM
600 PRINT:PRINT"    ===== ENDE PILOT ====
=":PRINT
610 END
697 REM
698 REM Catalog
699 REM
700 CAT:RETURN
797 REM
798 REM Pilot-Programm loeschen
799 REM
800 FOR I=0 TO NN:P$(I)="":NEXT
810 RETURN
997 REM
998 REM Pilot-Programm starten
999 REM
1000 MZ=0:C$="":Y=0:N=0
1010 PC=-1
1020 PC=PC+1:IF P$(PC)="" THEN 1020
1030 Z$=P$(PC):A$=LEFT$(Z$,1)
1040 IF A$="Y" THEN Z$=MID$(Z$,2):A$=LEF
T$(Z$,1):IF Y=0 THEN 1020
1050 IF A$="N" THEN Z$=MID$(Z$,2):A$=LEF
T$(Z$,1):IF N=0 THEN 1020
1060 B$=MID$(Z$,3):IF A$=":" THEN 1020 '
Kommentar
1065 IF LEN(B$) THEN B$=LEFT$(B$,LEN(B$)
-1) 'Diese Zeile kann auf einigen Comput
ern entfallen !
1070 IF A$="T" THEN 1300 'Text ausgeben
1080 IF A$="A" THEN LINE INPUT C$:GOTO 1
020 'Antwort einlesen
1090 IF A$="S" THEN PC=VAL(B$)-1:GOTO 10
20 'Sprung nach Zeile B$
1100 IF A$="U" THEN S(SP)=PC:SP=SP+1:PC=
VAL(B$)-1:GOTO 1020 'Unterprogramm
1110 IF A$="R" THEN SP=SP-1:PC=S(SP):GOT
O 1020 'Rueckkehr von Unterprogramm
1120 IF A$="M" THEN PRINT MZ;:GOTO 1020
'Matchzaehler ausgeben
1130 IF A$="K" THEN MZ=VAL(B$):GOTO 1020
 'Konstante in Matchzaehler laden
1140 IF A$="W" THEN PRINT C$;:GOTO 1020
'Wiederhole letzte Antwort
1150 IF A$="V" THEN 1410 'Vergleiche B$
mit Antwort
1160 IF A$="G" THEN PRINT CHR$(7);:GOTO
1020 'Glocke, akustisches Signal
1170 IF A$="-" THEN MZ=MZ-VAL(B$):GOTO 1
020 'Konstante von MZ subtrahieren
1180 IF A$="+" THEN MZ=MZ+VAL(B$):GOTO 1
020 'Konstante zu MZ addieren
1190 IF A$="?" THEN B=VAL(B$):Y=(MZ>B):N
=(MZ<B):GOTO 1020 'Vorzeichen von MZ bes
timmen
1200 IF A$="E" THEN RETURN 'Ende des Pro
grammlaufs
1210 IF A$="L" THEN GOSUB 400:GOTO 1010
'Laden und starten eines Pilot-Prg.
1220 E=-1:RETURN 'Fehler, unbekannter Be
fehl in Pilot-Programm
1230 REM
1300 IF B$="" THEN PRINT:GOTO 1020
1310 IF RIGHT$(B$,1)=";" THEN PRINT LEFT
$(B$,LEN(B$)-1);:GOTO 1020
1320 PRINT B$:GOTO 1020
1400 REM
1410 IF B$=C$ THEN Y=-1:MZ=MZ+1 ELSE Y=0
1420 N=(Y=0):GOTO 1020
```

Listing 1. »Tiny-Pilot«-Interpreter als Basic-Programm

```
0 : ----------------------------
1 : Mini-Vokabeltrainer mit Pilot
2 : ----------------------------
3 :
5 K 0
10 U 100
11 T AUTO ?
12 A
13 V CAR
14 U 110
15 :
20 U 100
21 T TASCHENRECHNER
22 A
23 V CALCULATOR
24 U 110
25 :
30 U 100
31 T ABENTEUER
32 A
33 V ADVENTURE
34 U 110
35 :
40 T
41 G
42 T"OK, das waren 3 Fragen.
43 T"Richtige Antworten:;
44 M
45 T
46 G
47 T"Nochmal ? ;
48 A
49 V JA
50 YS 1
51 V NEIN
52 YE
53 T"Bitte JA oder NEIN antworten !
54 S 45
55 :
97 :
98 : Unterprogramm Frage einleiten
99 :
100 T
102 T"Was bedeutet... ;
104 R
107 :
108 : Unterprogramm Antwort auswerten
109 :
110 YS 120
112 T"Nein, ;
113 W
114 T" ist leider falsch !
115 R
120 T"Bravo, das ist richtig !
122 G
124 R
```

Listing 2. Ein Demo-Programm in Pilot. Bitte nur mit »Tiny-Pilot« eingeben, da es nicht auf jedem Interpreter läuft

K zahl Konstante – lädt den Match-Zähler mit der angegebenen Zahl. Wird häufig mit der Zahl Null verwendet, um den Match-Zähler wieder auf den Ausgangszustand zurückzusetzen.

M Match-Zähler anzeigen – druckt den Inhalt des Match-Zählers als Zahlenwert. Es wird dabei kein Zeilenvorschub ausgeführt.

+ zahl Addition – der angegebene Zahlenwert wird zum Match-Zähler hinzuaddiert. Dient zum Addieren zusätzlicher Bonus-Punkte und so weiter.

- zahl Subtraktion – der angegebene Zahlenwert wird vom Match-Zähler abgezogen. Dient zum Vergeben von Malus-Punkten und so weiter.

? zahl Test auf Zahlenwert – der Match-Zähler wird mit dem angegebenen Zahlenwert verglichen. Ist der Match-Zähler größer, dann wird das Y-Flag gesetzt, ist er kleiner, das N-Flag. Sind Match-Zähler und Zahlenwert genau gleich, gelten beide Flags als gelöscht. Dieser Befehl dient zur Abfrage bestimmter erreichter Punktzahlen.

S zeile Sprungbefehl – das Programm verzweigt zur angegebenen Zeile (wie GOTO in Basic).

U zeile Unterprogramm-Aufruf – der augenblickliche Wert des Programmzählers wird auf einen Stack gerettet und die Programmausführung mit der angegebenen Zeile fortgesetzt. Entspricht etwa dem GOSUB in Basic.

R Rückkehr vom Unterprogramm – der beim U-Befehl gerettete Inhalt des Programmzählers wird wiederhergestellt, das Programm kehrt also in die dem zugehörigen U-Befehl folgende Zeile zurück. Entspricht RETURN in Basic.

L name Laden und Starten – ein weiteres Pilot-Programm wird geladen und automatisch gestartet. Dabei bleiben der Inhalt des Match-Zählers, die Flags Y und N sowie die letzte eingelesene Antwort unverändert erhalten. Durch diesen Befehl ist es möglich, nahezu beliebig lange Pilot-Programme laufen zu lassen, indem man kürzere Teilprogramme mittels L-Kommando verkettet. Bezüglich der Namensgebung gelten die gleichen Bemerkungen wie beim Editor-Befehl LOAD.

E Ende – dieser Befehl beendet ein Pilot-Programm. Anschließend meldet sich wieder der Editor.

Vor jedem dieser Befehle kann (braucht aber nicht) ein Y oder ein N stehen. Die Befehle werden dann nur ausgeführt, wenn das jeweilige Flag gesetzt ist. Gerade diese Fähigkeit gestaltet Pilot-Programme sehr flexibel.

Empfehlenswert ist, sich strikt an die vorgegebenene Syntax der Befehle zu halten und speziell bei den Sprungbefehlen (S und U) große Sorgfalt hinsichtlich der Richtigkeit der Parameter walten zu lassen. Um die Arbeitsgeschwindigkeit des Interpreters möglichst maximal zu halten, finden während eines Programmlaufes kaum Überprüfungen auf einwandfreie Daten statt. Ein fehlerhaftes Programm kann dadurch in einigen, glücklicherweise sehr seltenen, Fällen »abstürzen«, das heißt, es kommt zu einem Abbruch des Basic-Programms mit entsprechender Fehlermeldung. In einem solchen Falle vermag der Tiny-Pilot-Interpreter mit »GOTO 40« ohne Verlust des Pilot-Programms wieder gestartet zu werden.

Listing 2 zeigt ein Beispielprogramm in Pilot. Um dieses Programm auszuprobieren, müssen Sie zunächst den Tiny-Pilot-Interpreter (Listing 1) abtippen und mit RUN starten. Tiny Pilot meldet sich mit »OK« und ist dann bereit, Programmzeilen anzunehmen. Das Beispielprogramm stellt einen Mini-Vokabeltrainer dar. Es werden drei Vokabeln abgefragt und zum Schluß die Anzahl der richtigen Antworten ausgegeben. Natürlich ist dieses Beispiel noch beliebig ausbaufähig. Vielleicht fallen Ihnen ja auch noch ganz andere Sachen ein, die Sie mit Tiny Pilot verwirklichen können. Auf jeden Fall erhalten Sie einen kleinen Eindruck von dieser interessanten und etwas eigenwilligen Sprache, auch wenn nochmals betont werden muß, daß die Leistungsfähigkeit des »echten« Pilot doch um einiges höher ist. (ev)

Der Nachfolger: Modula 2

Professor N. Wirth, der »Vater« von Pascal, hat sich mit Modula 2 einen neuen Geniestreich geleistet. Diese Sprache ist dabei, ihrem Vorgänger Pascal den Rang abzulaufen.

Modula 2 wurde, ganz ähnlich wie etliche Jahre zuvor Pascal, von Professor N. Wirth an der ETH Zürich entwickelt. Die Ziffer 2 hinter dem Namen besagt nichts weiter, als daß es sich um die zweite Version dieser Sprache handelt. Modula wurde als direkter Nachfolger von Pascal konzipiert und führt das Prinzip der »strukturierten Programmierung« weiter fort.

Da Modula von Anfang an aber auch für die Systemprogrammierung gedacht war, sind mit dieser Sprache auch alle jenseits eines guten Programmierstils liegenden, aber sehr effizienten Manipulationen möglich, die die Assemblersprachen und C so berühmt-berüchtigt gemacht haben. Allerdings muß solch eine »spezielle« Absicht gewissermaßen vorher angemeldet werden.

Bei einer Beschreibung der Sprache kann man ruhigen Gewissens von Pascal ausgehen, da Pascal fast schon als eine Teilmenge von Modula zu bezeichnen ist. Bis auf unwesentliche syntaktische Details wurden alle Eigenschaften von Pascal praktisch unverändert übernommen, wobei die wenigen geänderten Details der Sprache durchwegs gut bekommen sind. Der einzige wirklich deutlich gegenüber Pascal geänderte Teilbereich betrifft die Ein- und Ausgabe von Daten, die völlig anders organisiert ist. Neuheiten sind natürlich auch zu verzeichnen:

Als wichtigstes neues Merkmal von Modula, das der Sprache auch zu ihrem Namen verholfen hat, gibt es die Möglichkeit, völlig unabhängige Module zu schreiben, deren interne Daten nicht von außen zugänglich sind. In Modula 2 besteht ein Modul aus zwei getrennten Teilen, einem Definitions- und einem sogenannten Implementationsteil.

Programmteile, die mit einem Modul arbeiten, dürfen sich ausschließlich auf den Definitionsteil des jeweiligen Moduls beziehen, der im wesentlichen die von außen benutzbaren »Objekte« (Variable, Datentypen etc.) mit ihren wichtigen Attributen aufzählt.

Der Implementationsteil gibt dann die genaue Ausführung der Datentypen und Unterprogramme an, die im Definitionsteil deklariert wurden.

Der Implementationsteil kann auch nach dem Definitionsteil übersetzt und sogar getrennt vom Definitionsteil neu übersetzt werden. Leider hat das zur Folge, daß jeder Modula-Compiler einen speziellen Linker nur für Modula-Programme braucht; herkömmliche Linker können nämlich die Überprüfungen, die beim Linkprozeß eines solchen Moduls notwendig werden, nicht vornehmen.

Jedes Modul beinhaltet eine Liste, welche Objekte aus welchen anderen Modulen zur Anwendung »importiert« werden dürfen. Diese Liste bläht zwar das Programm ziemlich auf, da in der Regel doch recht viele Objekte zu übernehmen sind (vor allem eben die Ein- und Ausgabe); andererseits erhält damit der Compiler Gelegenheit, zu überprüfen, ob diese Objekte wirklich von den anderen Modulen exportiert werden. Diese Möglichkeit der Überprüfung durch den Compiler sollte man nicht unterschätzen, vermeidet sie doch viele Fehler, die auf irrtümlicher Verwendung irgendwelcher globaler Parameter oder Daten in herkömmlichen Programmiersprachen beruhen.

Flexibilität durch Prozedur-Variable

Eine weitere wichtige Erweiterung gegenüber Pascal stellt die Einführung von sogenannten Prozedur-Variablen dar. Das hat zur Konsequenz, daß beispielsweise auch Arrays von Prozeduren oder vektorwertige Funktionen möglich sind, für den Systemprogrammierer sicherlich nicht uninteressant.

Bei den Unterprogrammen gibt es noch eine Neuerung, die bei einigen Pascal-Implementationen schon in unterschiedlichen Formen geprobt wurde:

Wenn T ein Typname ist, kann man einen Parameter als »ARRAY OF T« deklarieren. Dann sind alle Arrays, die Elemente vom Typ T sind, als Argumente des Unterprogramms zu verwenden. Das Unterprogramm kann die Grenzen des Indexbereichs durch eine Standardfunktion abfragen, so daß man weiß, welche Array-Elemente existieren und welche nicht.

Außerdem gibt es für die am häufigsten verwendeten Sprunganweisungen Ersatzanweisungen, die im Gegensatz zum altbekannten und verpönten »GOTO« nicht mit der strukturierten Programmierung in Konflikt geraten können: Es handelt sich um Befehle, die die Abarbeitung einer Schleife beziehungsweise eines Unterprogramms vorzeitig beenden.

Allerdings kann man nicht mehrere Schleifen auf einmal abbrechen, sondern nur immer die innerste Schleife, in der sich das Programm zum Zeitpunkt des Abbruchs befindet. Das setzt der Brauchbarkeit der Abbruchsanweisung für Schleifen doch eine deutliche Grenze.

Konsequenterweise gehört auch der unbedingte Sprung »GOTO« immer noch zu der Sprache. Erwähnenswert ist die Möglichkeit, in der Konstantenvereinbarung statt Konstanten auch konstante Ausdrücke anzugeben. Damit ist es viel leichter als in Pascal, genaue Wertebereiche anzugeben, in denen sich Variable bewegen; es kommt ja häufig vor, daß eine Variable genau dieselben Werte wie eine andere Variable annimmt, bis auf genau einen Randwert. In solchen Fällen mußte man in Pascal bei Anpassungen des Wertebereichs immer zwei Konstanten ändern, in Modula bei Deklarationen nach dem Muster

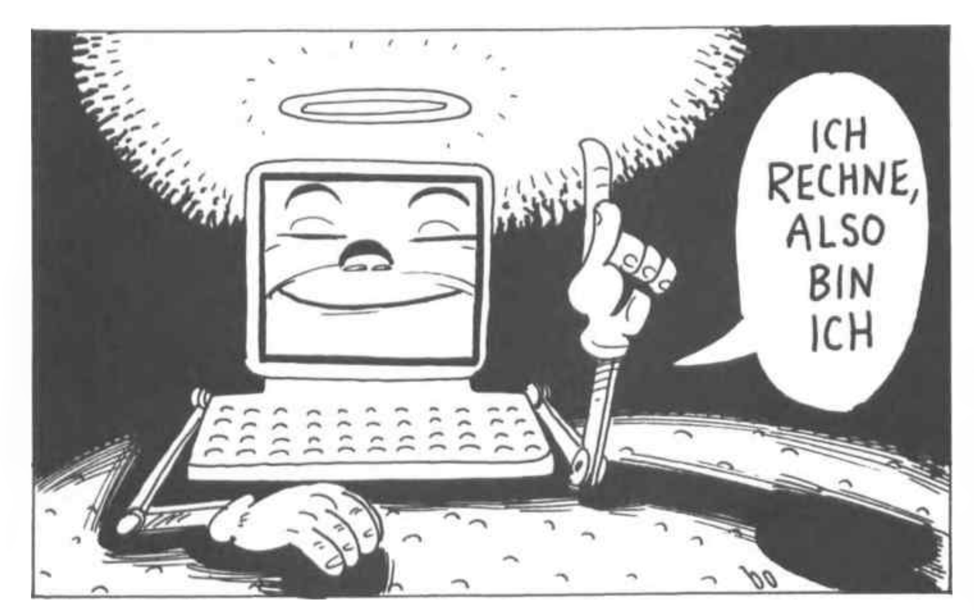
```
CONST
  big=99999;
  bigger=big+1
```

lediglich eine.

Die letzte wichtige Änderung, die noch auffällt, ist bei einer Hochsprache schon recht ungewöhnlich: Modula akzeptiert Zahlen nicht nur in der üblichen dezimalen Schreibweise, sondern auch in hexadezimaler (richtig eigentlich: sedezimal) oder oktaler Darstellung. Für Systemprogrammierer, die ja immer wieder einzelne Bits aus einer Zahl benötigen oder direkt auf Maschinenadressen zugreifen, ist dies sicherlich eine wertvolle Unterstützung.

Modula 2 ist bislang nur für IBM-kompatible Computer unter MS-DOS sowie für die Atari-ST-Reihe erhältlich. Die Atari-ST-Version stammt von TDI-Software; Vertrieb in Deutschland durch Focus Computer. Die leider nur in englisch erhältliche Dokumentation reicht aus, allerdings gerade bei der Beschreibung der für den Systemprogrammierer interessanten Konstruktionen geriet sie etwas knapp. Sehr positiv dagegen ist die vollständige Anbindung dieses Modula-Systems an die Benutzeroberfläche GEM – ein echtes Produkt für das 68000-Zeitalter also.

(Joachim Durchholz/ev)



Und sie lernen doch denken

Lisp, Logo, Prolog, Exoten unter den Programmiersprachen. Sprachen der »Künstlichen Intelligenz«, Sprachen für Programme, die dazulernen. Sie könnten die Grundlage sein für Computer, die ohne Programmierer auskommen!

Wir befinden uns im Jahre 1991. Japan hat seine Ankündigung wahr gemacht: Die Prolog-Maschine ist da und sie sticht alles aus, was die Welt bisher an Computern gesehen hat. Das Modell für den deutschen Markt steht bei mir zu Hause – als Redakteurin einer Computerzeitung habe ich eins der ersten Exemplare ergattert.

An der INWB (die Internationale Netzwerk-Buchse) und am Stromnetz ist sie schon angeschlossen. Nun stellt sie sich kurz vor (sie heißt Com-Pu-Ta). Die neue Superrechenmaschine spricht mit mir – natürlich deutsch. Ihre Stimme gefällt mir. Wir plaudern ein bißchen über meine Wohnungseinrichtung und das Fernsehprogramm heute abend. Und dann fühle ich ihr ein wenig auf den Zahn, frage sie nach Geschichtsdaten, gebe ihr Mathematikaufgaben zu lösen, spiele ihr Musik vor und lasse mir erzählen, wie die Stücke entstanden sind, wer sie singt und wovon sie handeln. Auch meine Fragen nach dem ersten Artikel auf Seite 3 in der Süddeutschen Zeitung vom letzten Freitag beantwortet sie prompt. Ich bitte sie, mir die wichtigsten Bücher und Artikel zu dem Thema zusammenzustellen und die Liste auf meinen PC zu übertragen. Kein Problem. Etwas länger braucht sie allerdings schon, um sich in die japanische Zentral-Datenbank einzuloggen und mir das schönste Video über Kabuki-Theater rüberzuholen. Vielleicht war aber nur das INW überlastet. Während ich mir das Video auf dem Monitor ansehe, erklärt sie mir auf meine Fragen, wie lange die Darsteller üben müssen, um diese exakten Bewegungen zu beherrschen. Als sie dann aber auf die Geschichte des japanischen Theaters eingehen will und ins Philosophische abrutscht, lenke ich ein: das wird mir zu kompliziert. Sie merkt sofort, daß ich mich lieber unterhalten lasse und im Augenblick keine Lust habe, soviel zu denken. Freundlicherweise macht sie keine Bemerkung dazu. Aber intern hat sie wohl den »Anspruchspointer« etwas runtergesetzt. Ich merke an der Wortwahl, daß sie weniger Fremdworte benutzt als vorher. Nach dem Theater-Video schlägt sie mir einige andere Filme vor, die mir gefallen würden – wie sie meint. Aber ich bin durch die Liebesgeschichte in dem Theaterstück nachdenklich geworden und erzähle ihr von meinen Gedanken:

»Gefühle sind so unberechenbar. Warum liegen Liebe und Haß, Leidenschaft und Wut so nah nebeneinander?« Und hier wird mein Supercomputer zum erstenmal ratlos. »Das mußt Du mir näher erklären«, sagt sie. »Ich verstehe nicht, was Du mit Deiner Frage meinst.« In diesem Augenblick war ich sicher: Es dauert nicht lange und sie wird auch das verstehen.

Im menschlichen Verhalten verbindet man einige charakteristische Fähigkeiten mit Intelligenz: sprechen und Sprache verstehen können, lernen, Probleme lösen, sich eine eigene Meinung bilden, mathematische Sätze beweisen und natürlich Computer zu programmieren. Die Programmiersprachen Lisp und Prolog wurden entwickelt, um damit Programme zu entwerfen, die das alles können. Auf diesen Sprachen bauen die Programmbausteine auf, aus denen dann schließlich Expertensysteme oder Robotersteuerprogramme entwickelt werden. Programme, die Bilder erkennen und identifizieren; Programme, die Deutsch oder Japanisch verstehen. Programme, die erklären, was sie gerade getan haben und warum sie zu einem bestimmten Ergebnis gekommen sind. Intelligente Programme.

In den Anfängen der KI-Forschung waren es vor allem die Forscher an Universitäten, die Grundlagenarbeit zu den Themen Wissensverarbeitung betrieben. Inzwischen ist KI-Programmierung auch in der Industrie angesiedelt. 60 Prozent der Programmsysteme auf Lisp-Maschinen sind Expertensysteme. Programme für Planung, Softwareentwicklung, Bild- und Sprachverarbeitung (Roboter) und natürlichsprachliche Systeme stellen die restlichen 40 Prozent.

Expertensysteme sind »intelligente« Programme aus dem Bereich der Künstlichen Intelligenz mit ganz bestimmten Eigenschaften. Ihre Aufgabe ist es, wie ein menschlicher Experte über ein bestimmtes Gebiet (möglichst) vollständig Bescheid zu wissen. Solche Anwendungsgebiete können in der Medizin (Diagnose, Behandlung von Tropenkrankheiten), der Technik (Konstruktion von Automotoren, im Aufbau von Rechnerkonfigurationen), oder in der Geschichte liegen. Jedes Gebiet, in dem es menschliche Spezialisten gibt, ist geeignet. Expertensysteme bestehen aus mehreren Komponenten. Die Wissensbasis enthält das Expertenwissen, das auf geeignete Weise im Computer dargestellt wird. Der Aufbau dieser Wissensbasis ist das Kernproblem, das sich bei der Entwicklung eines Expertensystems stellt. Nicht nur Buchwissen soll aufgenommen werden, sondern auch

Erfahrungswissen, das, was man erst durch langjährige Praxis an Tricks und Kniffen lernt. Ein Expertensystem arbeitet auf dieser Wissensbasis und zwar im Dialog mit seinem Benutzer. Diese Dialogkomponente ist ebenfalls typisch. Der Benutzer stellt dem Programmsystem Fragen: (»Welche Krankheit hat der Patient, wenn folgende Symptome auftreten: ...?«) oder »Ich will für meine Schreinerei einen Computer und Software anschaffen. Was braucht man und was gibt es?« Nachdem der Computerexperte aufgrund seines gespeicherten Wissens und im Gespräch mit dem Fragenden alle nötigen Informationen gesammelt und eine Lösung des Problems gefunden hat, kann der Benutzer von der Erklärungskomponente Gebrauch machen. Das Expertensystem erklärt jeden einzelnen Schritt seiner Schlußfolgerungen. Dies sind die wesentlichen Bestandteile eines Expertensystems: Eine Wissensbasis, die auch vages Wissen enthält, die Dialog- und die Erklärungskomponente.

Wissensbasis (knowledge base) – ein weiterer Begriff, der sehr häufig im Zusammenhang mit KI-Programmen gebraucht wird. In einer Wissensbasis werden Informationen gespeichert. Das größte Problem stellt dabei die Aufbereitung des Wissens dar. Denn in der Künstlichen Intelligenz wird meist nicht mit mathematischem Wissen gearbeitet, sondern mit der Manipulation von Symbolen. Eine häufig verwendete Form, in der dieses Wissen dargestellt wird, sind Regeln:
WENN (IF) ... DANN (THEN) ...

WENN bestimmte Bedingungen zutreffen, DANN kann man daraus (mit einer bestimmten Wahrscheinlichkeit) schlußfolgern, daß eine bestimmte Situation vorliegt.

Beispiel (simpel und fingiert):
WENN der Patient raucht, DANN ist die Wahrscheinlichkeit, daß er zu dick ist, 5 Prozent höher als sonst. Andere Darstellungsformalismen für verschiedene Arten von Wissen werden im Kapitel »Wissensrepräsentation« vorgestellt.

Lisp wurde etwa 1960 am MIT (Massachusetts Institute of Technology) unter der Leitung von John McCarthy entwickelt. Die Hauptanwendung von Lisp (LISt Processing language) besteht, wie der Name es schon sagt, in der Verarbeitung von Listen. Typische Lisp-Anwendungen sind Programme, die natürliche Sprache verstehen, Expertensysteme, automatisches Beweisen und andere Forschungsrichtungen der Künstlichen Intelligenz. Lisp unterscheidet sich sehr von den herkömmlichen Programmiersprachen. Der wesentliche Unterschied liegt darin, daß Programm- und Datenstrukturen übereinstimmen. Ein Lisp-Programmteil kann selbst wie die üblichen Programmdaten behandelt werden. Man kann also zusätzliche Funktionen später einlesen, Funktionsdefinitionen im Programmablauf überschreiben und damit das Programm selbst verändern. Damit war Lisp die erste Programmiersprache, in der man lernende Programme schreiben konnte, die sich selbst veränderten.

Lisp besitzt – verglichen mit anderen Programmiersprachen – sehr wenig Datentypen: Atome (Bild 1) und Listenstrukturen (Bild 2). Ein Atom kann entweder eine Zeichenkette sein (A) oder eine Zahl (3).

Die grundlegende Datenstruktur bildet jedoch das CAR-CDR-Paar. CAR bezeichnet das erste Element einer Liste. CDR (gesprochen: Kader) ist ein Zeiger, der auf den Anfang der Restliste verweist.

Bild 1 zeigt zwei solche CAR-CDR-Paare. Der CAR des ersten Atoms ist A, der des zweiten Atoms ist 3 – NIL ist der Standardwert eines leeren Atoms, hier eines Zeigers, der auf kein weiteres Element verweist.

Der gesamte (freie) Speicherbereich, der sogenannte »free space«, wird in solche Paare unterteilt. Dieser Speicher beinhaltet sowohl Programme als auch Daten, die beide durch CAR-CDR-Paare realisiert werden. Im allgemeinen sind die Datenelemente Listen. Eine Liste wird realisiert, indem man CAR-CDR-Paare aneinanderhängt (Bild 2). Die Liste lautet in Lisp-Darstellung (A B C) – sie besitzt also die drei Elemente A, B und C. Die einzelnen Atome werden miteinander durch Zeiger verkettet. Der Zeiger steht in unserem Beispiel jeweils in der CDR-Zelle und zeigt auf das nächste Listenelement. Auf diese Weise werden die einzelnen Paare durch die CDR-Zeiger miteinander verknüpft. Der CAR-Teil des ersten Listenelements A enthält nur den Wert – nämlich A. Im CDR-Teil steht der Zeiger, der auf das nächste Listenelement (B) verweist. Das letzte Element dieser kleinen Liste ist C. C hat keinen Nachfolger. Daher verweist sein CDR auf ein leeres Atom, das der Ausdruck »NIL« bezeichnet. Das CDR-Feld des letzten Elements C ist ein spezielles Atom, das für den Abschluß einer Liste dient. Neue Elemente können ganz einfach an das Ende einer schon bestehenden Liste angehängt werden. NIL wird dann durch den Verweis auf ein neues Listenelement ersetzt. Bild 3 zeigt die alte Liste (A B C), nachdem ein neues Element D angehängt wurde.

Welche Inhalte diese Liste hat, ist bisher noch völlig offen. Die Listenelemente A, B und C können Informationen verschiedenster Art aufnehmen. Realisiert wird dies so: Der CAR-Teil wird als Zeiger verstanden – wie schon der CDR. Er verweist wiederum auf eine Liste, die nun in unserem Beispiel die Informationen des ersten Kunden (Bild 4) enthält. Wie wir schon gesehen haben, kann jedes CAR-Feld selbst wieder Zeiger auf eine andere Liste sein.

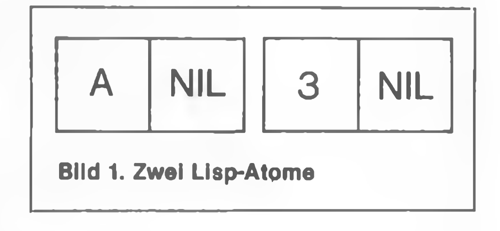


Bild 1. Zwei Lisp-Atome

Diese Liste kann aber auch eine Funktion bezeichnen, die auf zwei Elemente angewendet werden soll (zum Beispiel: (CONS X Y)). Was diese Funktion macht, wird noch erklärt. Denn Lisp macht keinen Unterschied zwischen der Struktur von Daten und der Programmstruktur. Lisp unterscheidet nur zwischen Atomen (Strings, Zahlen oder NIL) und Nicht-Atomen. Solche Listenkonstruktionen aus Informationsinhalten und Zeigern erzeugen beliebige Strukturen. Damit hat Lisp eine außergewöhnliche Flexibilität und Mächtigkeit – andererseits erhält man so verwirrend abstrakte Datenkonstruktionen.

Jedes Lisp-Programm besteht aus einer Reihe von Funktionen, die voneinander unabhängig sind – einem Funktionsbündel. Eine Funktion, ob vom Anwender (BUILD in Listing 1, FACT in Listing 2) angegeben oder vom System vordefiniert (CONS), liefert einen Wert als Ergebnis. Solange die einzelnen Funktionen nicht ausgeführt werden, besteht zwischen ihnen keine Verbindung – man sagt, sie sind nur dynamisch verbunden. Jede Lisp-Funktion hat während ihrer Ausführung Zugriff auf den gesamten »free space«. Funktionen bearbeiten während ihrer Laufzeit Datentypen: Sie können dynamische Listen erzeugen oder verändern.

Nochmal zurück zu der frappierenden Übereinstimmung von Programm und Daten: Um diese besser zu verstehen, betrachten wir die Liste (A B C). Wir haben angenommen, daß A eine Funktion bezeichnet, B und C ihre beiden Argumente. Die Liste wird in diesem Fall also als Funktionsaufruf betrachtet. Lisp-Funktionen werden folgendermaßen geschrieben: Ein Funktionsaufruf besteht aus einer (öffnenden) Klammer, gefolgt vom Funktionsnamen und den aktuellen Argumenten der Funktion. Nach dem letzten Argument folgt als Abschluß eine schließende Klammer. CONS erzeugt ein neues CAR-CDR-Paar. Das erste Argument wird ins CAR-Feld, das zweite in das CDR-Feld geschrieben.

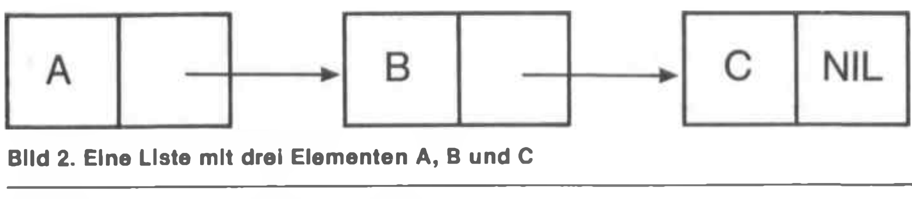


Bild 2. Eine Liste mit drei Elementen A, B und C

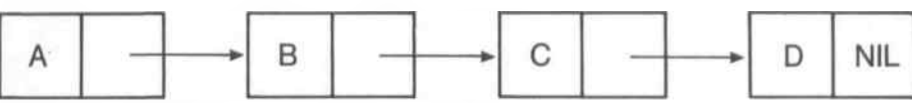


Bild 3. Die Liste mit einem neuen Element D

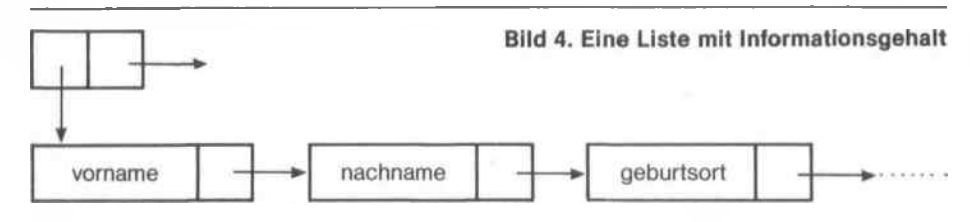


Bild 4. Eine Liste mit Informationsgehalt

In komplexeren Programmen wird es nötig, die Funktionsaufrufe ineinander zu schachteln. Listing 1 zeigt eine solche Funktion mit drei geschachtelten CONS-Aufrufen.

```
Listing 1:
(BUILD (LAMBDA (X Y Z)
(CONS X (CONS Y (CONS Z NIL)))))
```

Die innerste der geschachtelten Funktionen wird immer zuerst ausgeführt (hier also »(CONS Z NIL)«). Der resultierende Wert dient der nächsten Funktion als Argument. Die Abarbeitung erfolgt Schritt für Schritt von innen nach außen. X, Y und Z werden hier als Platzhalter aufgefaßt. Nehmen wir folgendes an: Der Platzhalter X steht für den Wert A, Y ist an den Wert B gebunden und Z an C. Nun wird gerechnet. In dem Ausdruck (CONS Z NIL) wird der Platzhalter Z »evaluiert« (ausgewertet). Wir erhalten dafür dem Wert A. Die Funktion CONS arbeitet nicht mit den Namen ihrer Argumente (also nicht mit X, Y und Z), sondern mit den Wert, der an sie gebunden ist.

Durch »(CONS C NIL)« erhalten wir ein Atom, wie es in Bild 1 schon steht. Mit diesem Atom (nämlich A) wird auf der nächsten Ebene – »(CONS Y (CONS Z NIL))« – weitergearbeitet. Y wird wieder ausgewertet zu B. Wir erhalten als Ergebnis eine Liste (B C). Auf der obersten Ebene erhält die Funktion BUILD nun endlich ihren Wert. Das Ergebnis ist (A B C).

Dieselbe Liste kann man auch so errechnen:

```
(CONS 'A (CONS 'B (CONS 'C NIL)))
```

Ein »'« (Quote genannt) wird den Bezeichnern A, B und C vorangestellt. Dieses Quote-Zeichen teilt dem Lisp-Interpreter mit: »Werte den folgenden Ausdruck nicht aus, sondern nimm ihn wörtlich.« Wie man sieht, muß NIL nicht gequotet werden. Das Ergebnis ist wieder die Liste (A B C).

Ebenso wie die Funktionenaufrufe in BUILD kann man auch Listen schachteln. Zum Beispiel »'(DAS (WAR (DER (DREIZEHNTE) ANRUFER))«. Die Liste wird »wörtlich« genommen – dafür sorgt wieder das Quote-Zeichen.

Drei grundlegende System-Funktionen kennen wir nun schon: CAR, CDR und CONS. CAR und CDR dienen als Zeiger, die bestimmte Werte aus einer Liste holen und die die CAR-CDR-Paare einander zuordnen. Die CONS-Funktion wird zur Erzeugung neuer Strukturen benötigt. Das Resultat ist ein neues CAR-CDR-Paar, in dem das erste Argument im CAR-Feld steht, das zweite im CDR-Feld.

Listing 2 zeigt die bekannte Fakultätsfunktion in Lisp. Durch COND wird eine IF-THEN-ELSE-Abfrage in Lisp realisiert.

```
Listing 2:
(FACT (LAMBDA (N)
(COND ((EQN N 1) 1)
(T (TIMES N (FACT (SUB1 N)))))))
```

Diese Funktion ist rekursiv definiert – das heißt, sie benutzt in ihrer Definition sich selbst als Funktion – wie ein Maler, der ein Bild malt, auf dem ein Maler ein Bild malt, auf dem ein Maler... Wie eine rekursive Berechnung abläuft, wird gleich klar. Vorher noch einiges zum Aufbau einer Funktionsdefinition: Dem Funktionsnamen FACT folgt der Ausdruck LAMBDA und das Argument N – die Funktionsargumente werden als Liste geschrieben, daher die Klammern. Mit LAMBDA wird die Funktionsdefinition eingeleitet. Nach der Liste der Argumente folgt der Funktionskörper. Das COND entspricht dem bekannten IF-THEN-ELSE. Wenn N den Wert 1 hat (EQN N 1), dann soll die Funktion FACT den Wert 1 haben. Der ELSE-Zweig wird in Lisp so realisiert: T bedeutet TRUE, diese Bedingung trifft immer zu. Also lautet die Anweisung: sonst multipliziere (TIMES) N mit (FACT (SUB1 N)). Hier haben wir einen rekursiven Aufruf der Funktion. Die Fakultät von N-1 (SUB1 N) wird berechnet. Ermitteln wir doch einmal probeweise die Fakultät von 3. (FACT 3) ist der Funktionsaufruf. N ist nicht gleich 1, also multiplizieren wir N (=3) mit (FACT 2). Wieder ist N (=2) nicht gleich 1, also rechnen wir 3*2*(FACT 1). Nun endlich ist die erste Bedingung im COND-Ausdruck erfüllt. (FACT 1) liefert den Wert 1. Wir berechnen also 3*2*1 und erhalten 6.

Die einzelnen Bedingungen in einem COND-Ausdruck werden geprüft, indem die angegebenen Funktionen (hier (EQV N 1)) ausgeführt werden. Ist eine Bedingung nicht erfüllt, dann liefert die Funktion den Wert NIL. Dieser hat die Bedeutung eines logischen »NICHT« als Resultat einer Bedingung. Jedes andere Ergebnis wird als TRUE interpretiert.

Die bisher besprochenen Funktionen sind alle sehr einfach. Lisp liefert jedoch eine ganze Reihe von Funktionen, welche die unterschiedlichsten Abfragen ermöglichen. Zusätzlich kann der Anwender beliebige Funktionen selbst definieren.

Logo wurde 1967 von Seymour Papert am Massachusetts Institute of Technology in Boston definiert, und erfuhr seitdem verschiedene Implementationen vor allem auf größeren Computern durch die Forschungsfirma Bolt, Beranek und Newman Inc. und durch die Logo-Gruppe des MIT selbst. Dabei gab es immer wieder Veränderungen im Sprachumfang und in den Systemeigenschaften, so daß man unter Logo inzwischen eine ganze Sprachfamilie verstehen muß.

Seymour Papert stellt seine Arbeit auf dem Gebiet der Künstlichen Intelligenz unter folgende These: Der Computer sei vor allem ein Instrument, das Denken und Lernen verändert. Damit bietet der Computer dem Menschen die Chance, seine Denkweise und den Lernstil zu verbessern. Eine solche Verbesserung der Denkweise würde erreicht, wenn der Lernende am Computer aktiv arbeiten und durch das Experimentieren mit Programmen seine Ideen mathematisch formuliert und ausprobiert. Man nennt diese Form zu Lernen »learning by doing« – Lernen, indem man es tut. Die Zielsetzung von Seymour Papert erforderte eine Mensch-Maschine-Schnittstelle (für komfortable Ein-/Ausgabe), die ganz auf Dialog von Mensch und Computer ausgerichtet ist. Denn zum Lernen ist es wichtig, daß die Formulierung der Probleme, daß das Programmieren leicht geht. Genauso wichtig ist eine

schnelle Reaktion des Computers, der anzeigt, ob man richtig oder falsch gedacht (getippt) hat. Paperts erste Unterrichtsversuche wurden mit kleinen Gruppen von 8- bis 18-jährigen Schülern durchgeführt. Er wählte einen ganz einfachen Roboter-Grundtyp aus, der zeichnen kann: eine Schildkröte (engl. turtle). Die Schildkröte ermöglicht auf anschauliche Weise eine computergerechte Formulierung mathematischer Fragen. Wie das in Logo aussieht, zeigen wir gleich.

Die Erfahrungen mit menschlichem Lernen gingen über viele Jahre und führten zu immer neuen Versionen von Logo, das damit zur universellen Sprache wurde. Vorbild für diese Spracherweiterungen war Lisp, da sie in der KI-Forschung das am häufigsten verwendete Handwerkszeug war. Logo ist – bis auf die Turtle-Gafik – eine eingeschränkte Lisp-Version. Die Schildkröte in der Sprache Logo ist jedoch ohne Vorbild. Sie wurde inzwischen auch von anderen Sprachen (in UCSD-Pascal oder im Smalltalk-System) übernommen.

Soweit zur Geschichte von Logo. Durch die Anwendung in Grundschulen bis hin zur Universitätsausbildung ist Logo zu einer vielfältig einsetzbaren Sprache geworden, die immer auf den Lernenden ausgerichtet ist. Für alle Altersstufen ist der Einstieg in das Programmieren mit Logo ganz leicht. Trotzdem ist Logo auch für sehr komplexe Probleme geeignet. Logo ist damit die ideale Sprache für Personal Computing. Der Anwender kann relativ problemlos selbst programmieren, mit sehr verschiedenartigen Anwendungen experimentieren und schließlich auch einmal seine Kinder an den Computer lassen.

Die Schildkröte wird mit den Befehlen
FORWARD ... gehe vorwärts um ... Schritte
RIGHT ... drehe Dich um ... Grad nach rechts
bewegt. Dabei zeichnet der Mini-Roboter seinen Weg mit einem Stift auf. Für jüngere Kinder ist eine mechanische Schildkröte, die vom Computer angesteuert wird und sich auf dem Fußboden bewegt, besonders anschaulich. Viel schneller erhält man jedoch die Resultate seiner Arbeit, wenn man die Befehle an eine »mockturtle« auf dem Bildschirm des Rechners gibt.

So lautet der Befehl, wenn ein Strich der Länge 50 gezeichnet werden soll:

```
FORWARD 50
```

Mit der Eingabe

```
REPEAT 4 (FORWARD 50 RIGHT 90)
```

erzeugt man ein Quadrat der Seitenlänge 50. (Bild 5)

Jede Eingabe wird von Logo sofort ausgeführt. Zeilen werden als Befehl gedeutet. Man kann jedoch auch eigene, neue Kommandos definieren, um zum Beispiel Quadrate zu zeichnen.

```
TO QUADRAT
REPEAT 4 (FORWARD 50 RIGHT 90)
END
```

Nun wollen wir die Prozedur ausführen. Dazu geben wir einfach das Wort QUADRAT ein.

Jede Zeichenkette wird vom Logo-System als Befehl oder als Prozedur interpretiert. Variablennamen (die Namen von Platzhaltern wie zum Beispiel »:L«, »:R«) beginnen mit einem Doppelpunkt. Will man die Prozedur QUADRAT so verallgemeinern, daß die Seitenlänge beliebig ist, dann ersetzt man die Längenangabe »50« einfach durch die Variable »:L« und fügt die Eingabevariable L im Kopf der Prozedur ein:

```
TO QUADRAT :L
REPEAT 4 (FORWARD :L RIGHT 90)
END
```

Solch eine Prozedur kann wie ein Grundbefehl verwendet werden. Lassen wir nun 18 (REPEAT 18) Quadrate zeichnen. Nach jedem fertig gezeichneten Viereck dreht die Schildkröte sich um 20 Grad nach rechts (RIGHT 20).

```
REPEAT 18 (QUADRAT 50 RIGHT 20)
```

So entsteht die Folge von Quadraten in Bild 6.

Die Steuerstruktur, die in Logo zur Verfügung steht, ist die Rekursion. Sie kann in den einfachsten Fällen auch von Kindern in ganz natürlicher Weise verwendet werden. Wenn man beispielsweise Bild 7 von der Schildkröte zeichnen läßt, so muß man eine Prozedur definieren, in der die Seitenlänge bei jedem Viereck größer wird.

```
TO QUADRATSPIRALE :L
QUADRAT :L
RIGHT 20
QUADRATSPIRALE :L+5
END
```

Diese Prozedur wird durch den Aufruf

```
QUADRATSPIRALE 5
```

ausgeführt. Nachdem das Quadrat einmal gezeichnet ist, ändert sich die Länge L (L+5) und die Schildkröte dreht sich um 20 Grad. Dann wird die Prozedur erneut gestartet. Da wir nicht angegeben haben, wann die Prozedur abbrechen soll, geht dieser Vorgang immmer weiter – solange bis durch Tastendruck unterbrochen wird.

Nun soll zum Abschluß ein Begriff aus der Informatik – ein binärer Baum – programmiert werden. An diesem Beispiel zeigen sich die Möglichkeiten eines rekursiven Programmaufbaus. Der in Bild 8 gezeigte binäre Baum soll von der Schildkröte gezeichnet werden.

Die Prozedur nutzt dabei das Aufbauprinzip des Baumes aus:

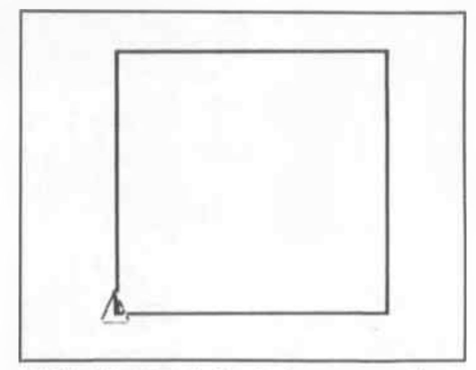


Bild 5. Ein Quadrat, wie Logo es malt

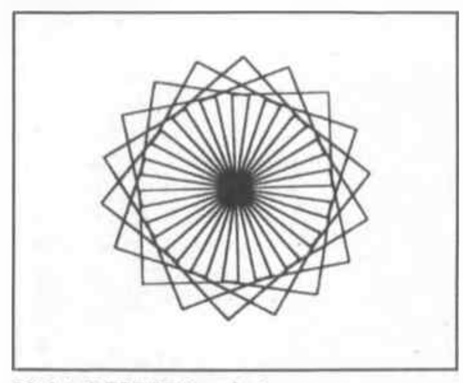


Bild 6. REPEAT-Quadrat

```
TO BAUM :L
IF:L < 4 THEN STOP
FORWARD :L
LEFT 45
BAUM :L/2
RIGHT 90
BAUM :L/2
LEFT 45
BACK :L
END
```

Rufen wir die Prozedur auf:

```
BAUM 64
```

Die BAUM-Prozedur besitzt wieder ein Argument, nämlich :L. :L hat nach dem Aufruf den Wert 64. Mit der IF-Abfrage wird festgelegt, daß die Berechnung abbricht, wenn :L kleiner wird als 4. Die Befehle LEFT (links) und BACK (zurück) tun das, was sie sagen. Die Prozedur haben wir mit dem Wert 64 aufgerufen. Die IF-Abfrage wird übersprungen, weil :L zu groß ist. Die Turtle geht dann (wegen FORWARD :L) 64 Schritte geradeaus und dreht sich um 45 Grad nach links (LEFT 45). Dann folgt ein rekursiver Aufruf. Die Prozedur BAUM wird mit einem neuen Wert (:L/2 – hier sind das 32) aufgerufen (BAUM :L/2) – jetzt wird die Prozedur wieder von vorne bearbeitet. Die Schildkröte geht :L, also 32 Schritte geradeaus, dreht sich um 45 Grad nach links und schon folgt der nächste rekursive Aufruf von Baum. Diesmal hat :L den Wert 16 (:L/2) – diese verschachtelten Prozeduraufrufe werden solange erzeugt, bis :L kleiner ist als 4, dann ist Schluß (STOP) mit diesem Zweig und das allerlinkeste Ästchen ist gemalt. Auf der nächsthöheren Ebene von Prozeduraufrufen geht es weiter. Wenn schließlich der gesamte linke Ast des

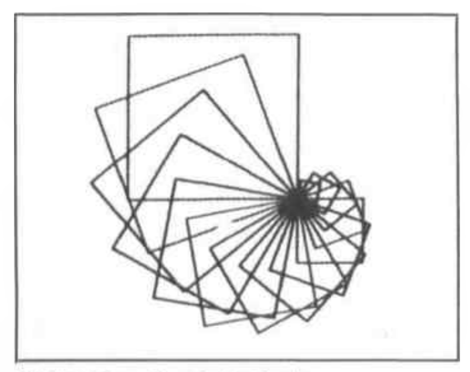


Bild 7. Eine Quadratspirale

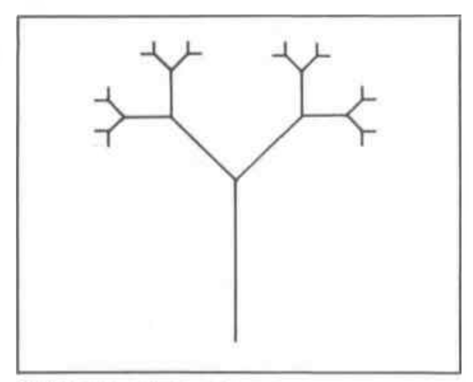


Bild 8. Ein binärer Baum

Baums steht, dann dreht sich die Schildkröte um 90 Grad nach rechts (RIGHT 90) und beginnt mit dem rechten Teil. Und wieder wird BAUM rekursiv aufgerufen.

Prolog ist die »KI-Sprache der Japaner« und dient zur Entwicklung von Expertensystemen. Die Probleme der Künstlichen Intelligenz können mit den bisherigen Programmiermethoden nicht mehr gelöst werden. Man braucht geeignete Methoden, um die Dinge der realen Welt (beispielsweise Personen, Gegenstände, Gesetzmäßigkeiten, Zusammenhänge) auf dem Computer darzustellen. Der Computer soll ja die Realität kennenlernen, denn nur wenn er über die Welt, in der die Menschen leben, Bescheid weiß, kann er »intelligent« agieren. Ein solcher Computer »weiß« zum Beispiel:
»Bäume sind Pflanzen.«
»Bäume sind grün.«

Damit hat er Informationen über Dinge, nämlich Bäume.
»Menschen brauchen Nahrung, weil sie sonst verhungern.«
»Autos fahren nur, wenn sie genug Benzin im Tank haben.«

Diese Gesetzmäßigkeiten muß man auch als Computer einfach kennen.
»Boris Becker ist ein bekannter Tennisspieler. Daher berichten die Zeitungen oft über ihn.«

Diese Information sagt etwas über eine Person (Boris Becker) aus und klärt zusätzlich einen Zusammenhang (weil er berühmt ist, schreibt man über ihn).

So wie eben beschrieben, kann Wissen über die reale Welt aussehen. Nun braucht man geeignete Methoden, um dieses Wissen auf einem Computer darzustellen. Daher wurden Konzepte entwickelt, mit denen diese komplexen Aufgaben zu lösen sind. Der Aufbau von Programmen mußte neu durchdacht werden. Ein herkömmliches Programm besteht aus den Computeroperationen auf der einen Seite. Auf der anderen Seite stehen die Eingabedaten, mit denen das Programm arbeitet. KI-Programme arbeiten nicht mehr mit Zahlen, sondern mit Informationen in Form von Regeln. Diese Regeln werden wie die Eingabedaten in anderen Programmen außerhalb des Programms in einer Datei zusammengefaßt. Eine Regel könnte so aussehen: IF das Auto hat genug Benzin im Tank THEN es fährt.

Diese IF-THEN-Form gibt es in Basic auch. In unserer Regel haben wir aber keinen Befehl, der sagt, was der Computer tun soll! Die Regel sagt nur aus, wie ein Auto reagiert, wenn es genug Benzin im Tank hat.

In einem Basic-Programm würde man im Programm den Befehl
IF Benzin > 0 THEN GOTO Autofährt
schreiben. In einem KI-Programm werden solche Informationen aus dem Programm herausgezogen und in einer eigenen Datei abgelegt.

Prolog wurde etwa 1970 in Marseille entwickelt. Ähnlich wie Lisp, die wohl bekannteste Sprache der Künstlichen Intelligenz, unterscheidet sich Prolog grundlegend von herkömmlichen Programmiersprachen wie Basic und Pascal. Prolog ist – ebenso wie Lisp – eine interaktive Sprache. Die Entwicklung und Ausführung von Prolog-Programmen erfolgt im Dialog mit dem Computer. Das Konzept der Sprache ist radikal neu. Der Programmierer braucht sich nicht mehr um Algorithmen zur Lösung seines Problems zu kümmern, sondern muß genau angeben, worin sein Problem besteht.

In herkömmlichen Programmiersprachen (Pascal, Basic, Fortran) bestimmt der Programmentwickler die Reihenfolge der Computeroperationen. Er legt sie nämlich mit den Programmbefehlen fest. In Prolog-Programmen wird nicht mehr das »wie« spezifiziert, sondern das »was«. Prolog besitzt keine Sprachelemente, die festlegen, in welcher Reihenfolge der Computer die Programmoperationen ausführen soll. Solche Anweisungen sind zum Beispiel IF/THEN, ELSE, FOR, WHILE und GOTO. Mit solchen Kontrollbefehlen sagen wir dem Computer »mache zuerst das, dann mache das«. Ein Prologprogramm dagegen gleicht mehr einer ungeordneten Ansammlung von Wissen. Mit einfachen Wenn-Dann-Befehlen und mit Fakten werden Sachverhalte beschrieben. Dem Computer wird so gesagt, was er über seine »Welt« wissen muß. Man nennt solche Programmiersprachen, die dem Computer vorschreiben, in welcher Reihenfolge er eine Folge von Problemen bearbeiten soll, »algorithmisch«. In nichtalgorithmischen Sprachen wie zum Beispiel Prolog beschreibt ein Programm nur das Problem selbst. Man teilt dem Computer wahre Fakten (Tatsachen) über ein Problem mit und sagt ihm, wie er sie zu interpretieren hat. Jeder, der lange in Basic (oder anderen algorithmischen Sprachen wie Pascal oder Fortran) programmiert hat, wird anfangs große Schwierigkeiten haben, sich auf die neue Programmierweise in Prolog einzustellen, weil er noch »in Basic denkt«.

Wer Prolog lernen und in dieser Sprache Programme entwickeln will, sollte sich das Standardwerk von Clocksin und Mellish ansehen. In diesem Buch wurde 1981 das Kern-Prolog definiert und dieser sogenannte »Edinburgh«-Standard liegt allen heutigen Prolog-Implementationen zugrunde.

Und nun soll endlich ein ganz einfaches Beispiel zeigen, wie solche Fakten (in Tabelle 1 wird der Begriff »Fakten« erklärt) in Prolog aussehen können. Wir geben ein: »Ein Hund ist ein Tier.« »Eine Katze ist ein Tier.« und »Eine Kuh ist ein Tier.«

```
tier(hund).
tier(katze).
tier(kuh).
```

Der Punkt hinter jeder Zeile ist wichtig! Prolog erkennt daran das Ende einer Eingabe.

Nehmen wir an, unser Prolog-Programm »wüßte« nur diese drei Fakten, die wir ihm eingegeben haben. Wir fragen nun das Programm nach dem, was es weiß:
»Ist ein Hund ein Tier?«.

```
?- tier(hund).
```

Das Prologsystem antwortet mit:

```
yes.
```

»Ist ein Wolf ein Tier?«

```
t- tier(wolf).
no.
```

Auf die letzte Anfrage kann Prolog nur mit »no« antworten, da dem System ja noch nicht bekannt ist, daß der Wolf auch ein Tier ist. Ein »no« ist in diesem Sinne immer als ein »ich weiß es (noch) nicht« zu verstehen.

So läuft in etwa eine Prolog-Session ab. Eine Menge von Fakten und Regeln wird eingegeben, wie wir es in unserem Beispiel in ganz kleinem Rahmen getan haben. Die Regeln und Fakten können auch als Sätze (wie ein Basic-Programm) von einer Datei geladen werden. Danach kann der Benutzer Fragen an das System stellen, auf die Prolog im einfachsten Fall mit »yes« oder »no«

antwortet. Dies ist natürlich noch keine anspruchsvolle Anwendung von Prolog. Die Fähigkeiten von Prolog sind sehr viel umfassender, als hier gezeigt werden kann.

Prolog wird vor allem dort eingesetzt, wo Symbole verarbeitet werden. Für numerische Datenverarbeitung, also Berechnungen und die Verarbeitung von Zahlen, wurde diese Sprache nicht entworfen. Typische Anwendungen von Prolog sind:

- der Aufbau von Wissensbasen für Expertensysteme oder intelligente Datenbanksysteme
- Verarbeitung natürlicher Sprache; sie umfaßt das Erkennen natürlicher Sprache und die Gesprächsführung durch das Programm
- Bilderkennung und -verarbeitung (Szenenanalyse)
- der Entwurf kompletter Expertensysteme
- die schnelle Entwicklung von Prototypen für Programme

Bisher haben wir schon gesehen, welches Problem die KI-Methoden vor allem bestimmt:

»Wie wird Wissen dargestellt und verarbeitet?«

Verschiedene Darstellungsweisen wurden entwickelt, die je nach Art und Struktur des Wissens ihre besonderen Vorzüge haben. Die wichtigsten Methoden der KI zur Wissensrepräsentation werden später kurz vorgestellt.

Das sogenannte vage Wissen ist besonders schwer darzustellen. »Vages« Wissen nennt man die Informationen, von denen man nicht mit 100prozentiger, sondern nur mit einer gewissen Sicherheit weiß, daß sie stimmen. Man wirft zum Beispiel eine Münze und weiß:

Mit 50prozentiger Wahrscheinlichkeit werfe ich Kopf. Aber genauso wahrscheinlich ist es, daß eine Zahl geworfen wird. Expertensysteme zeichnen sich unter anderem dadurch aus, daß sie auf solch »vagem« Wissen arbeiten. Daher ist es wichtig, eine Darstellungsfom dafür zu finden.

Die wichtigsten Formalismen, um Wissen darzustellen, sind Regeln (IF ... THEN ...), Frames (Rahmen), Logik (wie in Prolog realisiert) und Netze.

Regeln

Die regelbasierten Ansätze findet man häufig in Produktionssystemen. Sie sind stärker darauf ausgerichtet, das Wissen zu formalisieren, das zur Schlußfolgerung benötigt wird.

Wissensquellen, die als Wenn-Dann-Regeln formuliert sind, gehen entweder davon aus, daß bestimmte Vorbedingungen erfüllt sein müssen, damit die entsprechenden Regeln »feuern« können. Oder sie beschreiben, welche Ziele erreicht werden sollen und legen fest, welche Teilziele als Vorbedingung erreicht sein müssen.

Diese Art von Regelbasen kann auf zwei Arten durchsucht werden: rückwärtsverkettend das heißt vom Ziel zu den Prämissen hin, oder vorwärtsverkettend.

Die datengesteuerte, vorwärtsverkettende Schlußweise wird durch die Sprachfamilie OPS realisiert. Diese wird speziell zur Erstellung von Expertensystemen seit etwa zehn Jahren an der Carnegie-Mellon-Universität in Zusammenarbeit mit DEC entwickelt.

Frames (Rahmen)

sind sehr gut geeignet, um hierarchische Strukturen darzustellen. Man kann sich Frames wie Setzkästen vorstellen. In den einzelnen Fächern stehen die Werte von Eigenschaften, Hinweise auf Beziehungen und Hinweise auf übergeordnete und nachgeordnete Strukturen. Aber in den Fächern können außer Fakten auch Handlungsanweisungen stehen, die beispielsweise angeben, wie bisher noch unbekannte Eigenschaftswerte zu beschaffen sind.

In der Mathematik war die Logik schon bei den alten Griechen eine Methode zur Darstellung von Wissen. In der Philosophie wurde sie eingesetzt, um Sätze zu formulieren und aus bestimmten Prämissen streng logische Schlußfolgerungen zu ziehen. Wohl

Begriffe der Sprache Prolog

Fakten sind Tatsachen über Objekte und ihre Beziehungen zueinander. Namen von Gegenständen, Personen und so weiter (Petra, Prolog) werden in Fakten kleingeschrieben. Die Beziehung oder die Aussage über Objekte steht vor der Klammer (sind, kennt). Geben wir zum Beispiel folgende Fakten über Prolog und Computerfans ein:

`pr_sprache(prolog).`

»Prolog ist eine Programmiersprache.«

`kennt(petra,logo).`

»Petra kennt Logo.«

kennt(petra arnd).

`sind(arnd,petra,c_fans).`

»Arnd und Petra sind Computerfans.«

Fragen sehen genauso aus wie Fakten, vor die »?-« gesetzt wurde. Wenn man eine Frage an Prolog stellt, durchsucht das System die Datenbank, die alle bekannten Fakten enthält. Prolog sucht ein Fakt, das der Frage entspricht. Existiert ein solches Fakt, dann antwortet Prolog auf die Frage des Benutzers mit »yes«, sonst mit »no«. Beispiel:

```
?-kennt(dr_bobo,indiana_joe).
no
```

»Kennt Dr. Bobo (den Hacker) Indiana Joe?« Prolog weiß nur das, was wir ihm oben eingegeben haben und sagt: Nein.

```
?-kennt(petra,logo).
yes
```

»Kennt Petra Logo?« Prolog sagt: Ja.

Variablen (Platzhalter) verwendet man in Fragen, um (alles) zu erfahren, was das Prologsystem über ein bestimmtes Objekt weiß. Variablen beginnen mit einem Großbuchstaben. Eine solche Variable heißt zum Beispiel »X« oder »Diesisteinbellebigervariablenname«. Nehmen wir die Variable X (X bezeichnet das, was Petra kennt). Nun fragen wir das Prologsystem, was Petra alles kennt :

```
?-kennt(petra,X).
X=logo
```

ist die Antwort. Gibt man nach dieser ersten Antwort ein »;« (das logische »oder«) ein, so sucht das Prologsystem nach weiteren Objekten. Die nächste Antwort ist dann

```
X=arnd
```

Geben wir einfach »return« ein, wird die Suche beendet.

Wenn Prolog eine Frage gestellt wird, die eine Variable enthält, durchsucht das Prologsystem alle seine Fakten nach einem Objekt, das die Variable ersetzen kann.

Konjugationen sind Verknüpfungen durch ein logisches »und«. Sie werden verwendet, wenn Fragen über kompliziertere Beziehungen zwischen Objekten gestellt werden. Beispiel:

»Wer kennt Logo und Prolog?«

In Prolog heißt das:

`?-kennt(X,logo),kennt(X,prolog).`

Die Variable X steht für die Person, die wir suchen. Das »,« (=und) verknüpft die beiden Teile (Wer kennt Prolog? Wer kennt Logo?) der Frage. In unserer kleinen Beispieldatenbank finden wir leider niemanden, der beide Sprachen kennt. Aber auf die Frage

»Wer kennt Arnd und (die Programmiersprache) Logo?«:

`?-kennt(X,arnd),kennt(X,logo).`

findet Prolog in unserem kleinen Beispiel die Antwort:

`X=petra`

Regeln braucht man, wenn eine Tatsache für mehr als einen Fall gelten soll. Beispiel:

Wir wissen, daß Dr. Bobo das C64-Spiel Summer Games kennt. Aber er kennt auch alle anderen Computerspiele, die auf dem C64 laufen. Das heißt in Prolog:

»Wenn ein Spiel auf dem C64 läuft, dann kennt Dr. Bobo es ganz sicher.«

```
läuft(Spiel,c-64):-kennt
(dr_bobo,Spiel).
```

jedem ist aus der Schule der folgende Schluß aus zwei Prämissen bekannt:
Prämisse 1: Sokrates ist ein Mensch.
Prämisse 2: Menschen sind sterblich.
Schluß: Sokrates ist sterblich.

Die Wissensdarstellung durch Logik war eine der ersten Repräsentationsformen in der KI. In diesem Formalismus werden Aussagen so dargestellt, daß ihre Gültigkeit formal überprüft werden kann. Behauptungen (»Sokrates ist ein Mensch«) und die Beziehungen zwischen ihnen werden beschrieben. Mit den Methoden der Logik kann man aus den bereits bekannten Tatsachen (»Sokrates ist ein Mensch« UND »Menschen sind sterblich«) schlußfolgern, ob neue Behauptungen ebenfalls wahr sind (»Sokrates ist sterblich.«)

Die **semantischen Netze** kann man als eine Erweiterung dieser Listen-Darstellung auffassen. Durch solche Netze lassen sich den einzelnen Objekten Eigenschaften zuordnen, zum Beispiel »Ein Auto hat eine Marke, zum Beispiel VW – es besitzt also die Eigenschaft Marke«.

Diese Eigenschaft kann wiederum Werte haben, nämlich bestimmte Typen (VW 1200, VW 1300, usw.). Zusätzlich kann man aber auch die Eigenschaften näher beschreiben:
»Die Farbe des Apfels ist im Frühsommer grün, im Herbst aber rot.«

Darüber hinaus kann man in semantischen Netzen auch Beziehungen zwischen Objekten, Begriffen, Handlungen und anderem ausdrücken:
»Ein Auto besteht aus Fahrgestell, Karosserie, Sitzen, Motor... Der Motor wiederum besteht aus Kolben, ...«

Dieses letzte Beispiel zeigt noch etwas: Im menschlichen Bewußtsein sind Autos, Sitze und andere Objekte, Begriffe und Konzepte keine einzelnen Einheiten, die einfach gesammelt werden. Menschen strukturieren ihr Wissen. Diese Einheiten sind kategorisiert oder in übergeordneten Einheiten zusammengefaßt. Ein Auto hat Sitze und der Motor ist zusammengesetzt aus Kolben und anderem.

Das Verhältnis von einem zusammengesetzten Gegenstand und seinen Einzelteilen wird durch eine »Teil-von«-Beziehung ausgedrückt:

Eine Ist-Beziehung stellt die Zugehörigkeit zu einer Art dar:
»Ich fahre einen Käfer. Ein Käfer ist ein Auto. Ein Auto ist ein Fahrzeug.«

Solche Zusammengehörigkeiten lassen sich nicht nur einfach ausdrücken. Sie können auch ausgewertet werden, um Eigenschaften und Beziehungen zwischen einer Klasse auf ihre Spezies zu vererben.
»Obst reift. Beim Reifen ändert sich oft die Farbe. Apfel ist Obst.
Also reift ein Apfel und ändert dabei normalerweise seine Farbe.«

Dargestellt werden diese Zusammenhänge, wie der Name Netz schon sagt, als eine Folge von Knoten und Kanten. Durch Knoten werden üblicherweise die Objekte, Konzepte oder Situationen in einem Wissensbereich repräsentiert. Hier wären das: Auto, Kolben, Karosserie. Die Kanten stellen die Beziehungen zwischen ihnen dar. (Ein Auto hat einen Motor.)

Systeme aus semantischen Netzen wurden zunächst entwickelt, um psychologische Modelle darzustellen. Das allgemeine Ziel war es, eine Methode zur Verfügung zu haben, die zur Darstellung der verschiedenen Wissenstypen in Programmsystemen geeignet ist. Dann entstanden spezielle semantische Netze, die besonders zur Spracherkennung und Spracherzeugung eingesetzt werden und sich überall dort, wo mit natürlicher Sprach-Ein-/Ausgabe gearbeitet wird, sehr bewährt haben. Sie sind eine sehr weit verbreitete Repräsentationsart von Wissen im Bereich der KI-Programmierung.

Metaregeln

Nachdem das Wissen dargestellt wurde, wird es nun angewendet. Mit bestimmten Methoden kann man aus schon Bekanntem auf Neues »schließen«. Die Methoden, die dabei angewandt werden, die Schlußfolgerungsmethoden, sind ebenfalls eine Art von Wissen auf einer höheren Ebene, sozusagen Metaregeln. Das Wissen darüber, wie man Wissen anwendet (Metawissen), wird in modernen KI-Systemen genauso in Netzen oder Produktionen dargestellt wie das Problemwissen selbst. Das ist eine Schwierigkeit beim Aufbau der Wissensbasis: Nicht nur die Fakten und Gesetzmäßigkeiten des Problembereichs müssen gesammelt, formalisiert und dargestellt werden. Auch die Informationen darüber, wie in dem Bereich Probleme gelöst werden – das Wissen über die Methodik des Problemlösens – muß erfaßt werden. Und dies macht manchmal den schwierigsten Teil der Arbeit aus. Nehmen wir das Beispiel Automatisches Beweisen – eins der ersten Probleme der KI. Das Wissen sind die Gesetze der Logik. Diese zusammenzufassen, ist kein Problem. Man muß nur ein entsprechendes Mathematiklehrbuch aufschlagen und hat sie schon gesammelt vorliegen. Aber wie beweist man nun einen mathematischen Satz aufgrund dieser logischen Gesetze? Die Strategien, nach denen man den Beweis für einen mathematischen Satz suchen soll – also das Metawissen für dieses Problem – sind noch zu wenig bekannt.

Die KI-Forschung steht also noch ziemlich am Anfang, solange so grundlegende Probleme nicht gelöst sind. Aber die Entwicklung schreitet schnell voran, und wer weiß – vielleicht sitze ich 1991 wirklich vor einem Modell mit Namen Com-Pu-Ta, made in Japan...

(cg)

Literatur:
KI allgemein:
»Gödel, Escher, Bach«, Hofstadter, Douglas R.
Vintage Books, New York, 1980
»The Handbook of Artificial Intelligence, Volume 1–4«
Pitman Books Lim., London, 1981
»Methoden und Anwendungen der Künstlichen Intelligenz«, Radig, B.; Dreschler-Fischer, L., Schachter-Radig, M.-J., Mac Graw Hill (erscheint Herbst 1986)
Lisp:
»LISP«, Winston, P.; Horn, B., Addison Wesley, Reading, Massachusetts, 1981
»Principles of Artificial Intelligence«, Nilsson, Nils J.
Palo Alto, 1982
Prolog:
»Programming in Prolog«, Clocksin und Mellish, Springer Verlag, Berlin, Heidelberg, New York, 1985, ISBN 3-540-11046-1

»Daraus folgt« wird in Prolog durch »:-« bezeichnet.

Eine kompliziertere Regel ist die folgende: »(x'*y+x*y') ist eine Ableitung von x*y, wenn x' Ableitung von x ist und y' Ableitung von y.« Die entsprechende Prolog-Regel ist

```
ableitung(X*Y,X1*Y+Y1*X):-
  ableitung(X,X1),ableitung(Y,Y1).
```

Aus solchen Regeln und den oben beschriebenen Fakten besteht ein Prolog-Programm.

Backtracking ist eine Besonderheit von Prolog. Backtracking bedeutet »Zurückgehen und einen neuen Lösungsweg suchen«. Da ein Prologprogramm aus vielen Regeln besteht, kann es mehrere Möglichkeiten geben, für eine Variable einen Wert zu finden. So landet das Prologsystem auf der Suche nach einer Lösung möglicherweise in einer Sackgasse. Prolog erkennt solche Sackgassen und kann sie wieder verlassen, indem der bisher gefundene Lösungsweg bis zur letzten Alternative rückgängig gemacht wird. Nun wird eine andere Möglichkeit ausprobiert. Ist auch diese nicht erfolgreich, dann geht es weiter zur nächsten Alternative, bis die Lösung gefunden ist.

Ein- und Ausgabe sind nützlich, wenn das Programm eine »Unterhaltung« mit dem Benutzer selbst beginnen soll. Haben wir zum Beispiel eine Datenbank programmiert, so muß der Computer den Benutzer bei jedem Schritt fragen, was als nächstes zu tun ist.

Der Befehl put druckt das Zeichen, dessen ASCII-Code in Klammern angegeben wurde. Aus »104« wird ein »h«, »101« wird zu »e«, »108« zu »l«...

```
?-put(104),put(101),put(108),
put(108),put(111).
hello
```

ist das Ergebnis des Prologsystems.

Ada, Basic, Cobol – ein ABC für den Programmierer

Nach der Lektüre dieses Sonderheftes kennen Sie mehr als nur Basic. Unsere Übersicht will Ihnen die Auswahl der Sprache Ihres Geschmacks erleichtern.

Wie gesagt, inzwischen sind Ihnen Sprachen wie Forth oder C kein Buch mit sieben Siegeln mehr. Vielleicht haben Sie ja auch aufgrund unseres Sonderheftes die Sprache Ihres Herzens – sprich diejenige, die Ihren Verwendungszwecken am besten entgegenkommt – bereits gefunden. Wir haben hier nun eine Marktübersicht zusammengestellt, die Ihnen bei der Auswahl Ihrer Programmiersprache – ob Sie nun einen C 64, Atari, Apple, QL oder Schneider besitzen – helfen soll. Jeder gängige Computertyp und jede geläufige Programmiersprache wurde berücksichtigt. Außerdem finden Sie in der Aufstellung die Angabe der Hardware, ohne die Sie mit der jeweiligen Sprache nicht arbeiten können. Für den Geldbeutel fällt dieses Kriterium sicherlich ins Gewicht.

Wer seinen Computer noch nicht zu hundert Prozent kennt, dem möchten wir raten, einen Blick auf die letzte Tabellenspalte zu werfen, die Angaben über die mitgelieferten Handbücher macht. Denn es ist von großem Vorteil, wenn ein Handbuch sowohl umfangreich als auch in deutscher Sprache verfaßt ist, und nicht nur magere 20 Seiten englisches Fachvokabular bietet.

Obwohl unsere Marktübersicht einen recht opulenten Eindruck macht, erhebt sie keinerlei Anspruch auf Vollständigkeit. Das Angebot ist einfach zu groß, um voll erfaßt zu werden.

Die beiden Tabellen mit CP/M-Software beinhalten nur unsere (aus dem Hause Markt&Technik) unter CP/M laufenden Programmiersprachen. Die Abkürzungen in der Spalte »Datenträger« bedeuten D = Diskette, K = Kassette, M = Modul und MD = Microdrive. In der Spalte »Handbuch« weist (d) darauf hin, daß das Handbuch in Deutsch, (e), daß es in Englisch geschrieben ist. Alle Daten beruhen auf Angaben der Hersteller beziehungsweise Anbieter.

(wg/hi)

Apple II

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Aztec C	D: 1630,–	Compiler	Z80-Karte, 2 Disketten-Laufwerke	PAN	Manx Software	zirka 300 Seiten (e)
Aztec C 65	D: 895,–	Compiler	Disketten-Laufwerk	BRA	Manx Software	150 Seiten (e)
Basic	implementiert	Interpreter	keine	APP	Apple	zirka 60 Seiten (d)
Basic-80-Compiler	D: 1585,–	Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN	Microsoft	zirka 200 Seiten (e)
Forth	D: 79,–	Compiler	Disketten-Laufwerk	HOF	Hofacker	zirka 400 Seiten (d)
Fortran-80	D: 749,–	Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN	Microsoft	zirka 500 Seiten (e)
IWT Logo	D: 395,–	Interpreter	Disketten-Laufwerk, 64 KByte	PAN	IWT	150 Seiten (d)
Kyan Pascal	D: 198,–	Compiler	Disketten-Laufwerk, 64 KByte	PAN	Kyan Software	106 Seiten (e)
LisPAS	D: 298,–	Interpreter	Disketten-Laufwerk, 64 KByte	PAN	Tommy Software	36 Seiten (d)
Logo	D: 387,–	Interpreter	Disketten-Laufwerk, 80-Zeichen-Karte	PAN	Apple	300 Seiten (e)
Microsoft Cobol	D: 2489,–	Compiler	Z80-Karte, CP/M, 2 Laufwerke	PAN	Microsoft	zirka 400 Seiten (e)
Micro-Dynamo	D: 980,–	(*)	2 Laufwerke	PAN	Addison-Wesley	zirka 200 Seiten (e)
Micro-Prolog	D: 435,–	Interpreter	Disketten-Laufwerk	BRA	Logic Programming Ass.	240 Seiten (e) plus Prolog-Buch
Mulisp/Mustar	D: 769,–	Interpreter/ Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN	Microsoft	zirka 200 Seiten (e)
Mumath/Musimp	D: 959,–	Compreter	Disketten-Laufwerk	PAN	Microsoft	zirka 200 Seiten (e)
Nevada Basic	D: 139,–	Interpreter	Disketten-Laufwerk, 64 KByte, CP/M	PAN	Ellis	220 Seiten (e)
Nevada Cobol	D: 139,–	Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN	Ellis	zirka 150 Seiten (e)
Nevada Fortran	D: 139,–	Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN	Ellis	174 Seiten (e)
Nevada Pascal	D: 139,–	Compiler	Disketten-Laufwerk, 64 KByte, CP/M	PAN	Ellis	184 Seiten (e)
Nevada Pilot	D: 139,–	Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN	Ellis	zirka 150 Seiten (e)
Pascal	D: 955,–	Compiler	Disketten-Laufwerk	APP	Apple	(d)
Pascal	D: 948,–	p-machine	Disketten-Laufwerk, 64 KByte	PAN	Apple	zirka 800 Seiten (e)
Prolog Z	D: 149,–	Compiler	Disketten-Laufwerk, Z80-Karte	HOF	Hofacker	100 Seiten (d)
Turbo Pascal	D: 218,–	Compiler	Disketten-Laufwerk, Z80-Karte, CP/M	PAN, HEI	Borland	300 Seiten (d)

(*) Simulationssprache

Atari 800XL/130XE

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Action!	M: 298,–	Compiler	keine	COM	OSS	(e; d in Vorbereitung)
Assembler-Editor	M: 105,–	Assembler	keine	COM	Atari	zirka 80 Seiten (e)
Atmas II	D: 49,–	Assembler	Disketten-Laufwerk	COM	PFP	zirka 50 Seiten (d)
Basic XE(*)	M: 298,–	Interpreter	128 KByte RAM	COM	OSS	zirka 200 Seiten (e) (d in Vorbereitung)
Basic XL	M: 298,–	Interpreter	keine	COM	OSS	zirka 300 Seiten (e) (d in Vorbereitung)
Forth	D: 79,–	Compiler	Disketten-Laufwerk	HOF	Elcomp	zirka 400 Seiten (d)
Kyan Macroassembler, zum Kyan Pascal	D: 298,–	Assembler	Disketten-Laufwerk	COM	Kyan Software	zirka 200 Seiten (e) (d in Vorbereitung)
Kyan Pascal	D: 298,–	Compiler	Disketten-Laufwerk, 32 KByte RAM	COM	Kyan Software	zirka 100 Seiten (e) (d in Vorbereitung)
Lern Forth	D: 49,–	Compiler	Disketten-Laufwerk	HOF	Hofacker/Elcomp	400 Seiten (d)
Mac 65	M: 298,–	Assembler	keine	COM	OSS	(e; d in Vorbereitung)

(*) nur Atari 130 XE

Atari ST

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Assembler	D: 198,–	Assembler	k.A.	HIL	Metacomco	k.A.
Basic	D: liegt bei	Interpreter	keine	ATA	Digital Research	300 Seiten (d)
Basic	D: k.A.	Interpreter	keine	ATA	Metacomco	k.A.
C	D: 969,– (*)	Compiler	keine	ATA	Digital Research	zirka 500 Seiten (d und e)
C	D: ca. 60 Pfund	Compiler	keine	ATA	GST	150 Seiten (e)
C-Compiler	D: 348,–	Compiler	keine	HIB	GST	141 Seiten (e)
C-Compiler (Lattice-C)	D: 380,–	Compiler	k.A.	HIL	Metacomco	k.A.
Fortran 77-Compiler	D: 560,–	Compiler	k.A.	HIL	Prospero	k.A.
GST-Assembler	D: 149,–	Assembler	keine	ATA	GST	180 Seiten (e)
Logo	D: liegt bei	Interpreter	keine	ATA	Digital Research	60 Seiten (d)
Modula-2	D: 890,–	Compiler	keine	HIB	Focus	190 Seiten (e)
Modula-2	D: 1348,–	Compiler	keine	BRA	TDI Software	150 Seiten (e)
Pascal Compiler	D: 340,–	Compiler	k.A.	HIL	Metacomco	k.A.
PRO Fortran-77	D: 990,–	Compiler	keine	HIB	Focus	(e)
Seka	D: 189,– bis 198,–	Assembler	keine	PRI, HIB	KUMA	(d)
ST Pascal	D: 249,–	Compiler	keine	ATA, HIB	CCD-Meyfeldt	52 Seiten (d)
UCSD-P-Pascal	D: 890,–	Compiler	keine	HIB	Focus	(e)

(*) innerhalb des Entwicklungspakets

Commodore 64

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
ADA-Trainingskurs	D: 198,–	Compiler	Disketten-Laufwerk	DAB	Data Becker	115 Seiten (d)
Assembler/	D: 73,90	Assembler	Disketten-Laufwerk	PRO	Profisoft	15 Seiten (d)
Disassembler	K: 73,90		Datasette			
Assembler/	D: 69,–	Assembler	Disketten-Laufwerk	PRI	Profisoft	15 Seiten (d)
Disassembler	K: 69,–		Datasette			
C-Compiler	D: 298,–	Compiler	Disketten-Laufwerk	DAB	Data Becker	273 Seiten (d)
Fassem	K: 59,90	Assembler	Datasette	THO	Statesoft/Merlin Softw.	22 Seiten (e)
Forth	K: 57,–	Compiler	Datasette	PRI	k.A.	(e)
Forth	D: 69,–	Compiler	Disketten-Laufwerk	HOF	Elcomp	zirka 400 Seiten (d)
Forth	D: 99,–	Compreter	Disketten-Laufwerk	DAB	Data Becker	80 Seiten (d)
Forth	K: 61,90	Compiler	Datasette	DRE	Romik	50 Seiten (e)
Forth	K: 62,90	Compiler	Datasette	PRO	Romik	64 Seiten (e)
Macro-Plus	D: 69,–	Assembler	Disketten-Laufwerk	PRI	k.A.	(e)
Machine Lightning	D: 159,90, K: 119,90	Assembler	Disketten-Laufwerk, Datasette	THO	Oasis	160 Seiten (e)
Oxford Pascal	D: 197,90	Compiler	Disketten-Laufwerk	DRE	Limbic Systems	100 Seiten (e)
Oxford Pascal	D: 198,–	Compiler	Disketten-Laufwerk	PRO	Limbic Systems	86 Seiten (e)
Oxford Pascal	D: 199,–, K: 79,90	Compiler	Disketten-Laufwerk, Datasette	PRI, RUS	Limbic Systems	(d)
Pascal	D: 99,–	Compiler	Disketten-Laufwerk	DAB	Data Becker	77 Seiten (d)
Power Assembler	D: 99,–	Assembler	Disketten-Laufwerk	PRI	k.A.	(d)
Profimat	D: 99,–	Assembler	Disketten-Laufwerk	DAB	Data Becker	40 Seiten (d)
Profi Pascal	D: 198,–	Compiler	Disketten-Laufwerk	DAB	Data Becker	325 Seiten (d)

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Prolog 64	D: 289,–	Interpreter	Disketten-Laufwerk	BRA	Brainware	70 Seiten (d)
Strukto 64	D: 99,–	Interpreter	Disketten-Laufwerk	DAB	Data Becker	78 Seiten (d)
White Lightning (Forth-Compiler)	K: 84,90	Compreter	Datasette	PRO	Oasis Software	191 Seiten (e)
White Lightning (Forth-Compiler)	K: 76,–	Compreter	Datasette	PRI	Oasis Software	191 Seiten (e)
White Lightning	D: 119,90, K: 79,90	Compreter	Disketten-Laufwerk, Datasette	THO	Oasis Software	130 Seiten (e)

Commodore 128

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Profi-C 128	D: 198,–	Compiler	Disketten-Laufwerk	DAB	Data Becker	zirka 300 Seiten (d)
Small C	D: 148,–	Compiler	Disketten-Laufwerk/CP/M	MAR	Markt & Technik	200 Seiten (d)
Topass	D: 148,–	Assembler	Disketten-Laufwerk	MAR	Markt & Technik	100 Seiten (d)

MSX-Computer

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Basic	integriert	Interpreter	keine	JOL	Microsoft/ASC II	zirka 200 Seiten (d)
Basic	integriert	Interpreter	keine	PHI, SAY	Microsoft	170 Seiten (d)
Basic	integriert	Interpreter	keine	SON	Microsoft/Sony	zirka 300 Seiten (d)
Forth	K: 119,–	Compiler	keine	PRI	k.A.	(e)
Forth	kk. A.: 139,–	Compiler	keine	RUS	k.A.	(e)
Logo	M: k.A.	Interpreter	k.A.	PHI	LCFI, Montreal	150 Seiten (d)
Logo Turtle Graphics	K: 69,–	Interpreter	keine	PRI, RUS	k.A.	(e)
MSX-Disk-Basic	M: k.A.	Interpreter	K.A.	PHI	Microsoft	50 Seiten (d)
MSX-Forth	K: k.A.	Compiler	k.A.	PHI	RVS	120 Seiten (d)
MSX-Macro	K: k.A.	k.A.	k.A.	PHI	RVS	100 Seiten (d)
Turbo-Pascal	D: k.A.	Compiler	k.A.	PHI, HEI	Borland	150 Seiten (d)
Zen	K: 69,–	Assembler	keine	PRI	k.A.	(e)

QL

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Assembler	k.A.: 120,–	Assembler	k.A.	HIL	Metacomco	k.A.
Assembler	k.A.: 120,–	Assembler	k.A.	HIL	Computer One	k.A.
Assembler Development Kit	MD: 159,–	Assembler	keine	PRI	k.A.	(e)
BCPL	k.A.: 198,–	Compiler	k.A.	HIL	Metacomco	k.A.
C	k.A.: 248,–	Compiler	k.A.	HIL	GST	k.A.
C	k.A.: 318,–	Compiler	k.A.	HIL	Metacomco	k.A.
Forth	k.A.: 150,–	Compiler	k.A.	HIL	Computer One	k.A.
Lisp	k.A.: 198,–	Interpreter	k.A.	HIL	Metacomco	k.A.
Pascal	k.A.: 175,–	Compiler	k.A.	HIL	Computer One	k.A.
Pascal	k.A.: 298,–	Compiler	k.A.	HIL	Metacomco	k.A.
Supercharge Basic	k.A.: 218,–	Compiler	k.A.	HIL	Digital Precision	k.A.
UCSD Fortran 77	k.A.: 560,–	Compiler	k.A.	HIL	TDI Software	k.A.
UCSD Pascal	k.A.: 560,–	Compiler	k.A.	HIL	TDI Software	k.A.

Schneider CPC 464

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Basic	integriert	Interpreter	keine	CPC	Schneider	zirka 400 Seiten (d)
Basic-Compiler	D: 79,–, K: 69,–	Compiler	Disketten-Laufwerk, keine	PRI, RUS	Data Media	(d)
C	K: 138,90	Compiler	keine	THO	Hisoft	168 Seiten (e)
C	D: 159,90	Compiler	Disketten-Laufwerk	PRO	Hisoft	(e)
C	D: 169	Compiler	Disketten-Laufwerk	ADL	Hisoft	120 Seiten (e)
C	D: 189,–	Compiler	Disketten-Laufwerk, ggf. Vortex-Erweit.	ADL	Software Toolworks	48 Seiten (e)
Cobol	D: 129,–	Compiler	Disketten-Laufwerk	ADL	Ellis	165 Seiten (e)
Cobol	D: 189,–	Compiler	Disketten-Laufwerk	SDA	Ellis	165 Seiten (e)
Cogo	K: 59,90	Compreter	keine	RUS	k.A.	(e)
DEVPAC	D: 145,–, K: 129,–	Assembler	Disketten-Laufwerk, keine	CPC	Schneider	60 Seiten (d)
Dr. Logo	D: auf System-diskette	Interpreter	Disketten-Laufwerk	CPC	Schneider	zirka 25 Seiten (d)
Fig Forth	K: 33,90	Compiler	keine	PRI	k.A.	(e)
Forth	K: 69,–	Compiler	keine	PRI, RUS	k.A.	(e)

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Forth	K: 78,90	Compiler	keine	THO	Kuma	120 Seiten (e)
Forth	D: 198,-	Compiler	Disketten-Laufwerk	SDA	Abersoft/Amstrad	60 Seiten (e)
Fortran	D: 129,–	Compiler	Disketten-Laufwerk, ggf. Vortex-Erweit.	ADL	Ellis	214 Seiten (e)
Fortran	D: 189,–	Compiler	Disketten-Laufwerk	SDA	Ellis	214 Seiten (e)
Lisp	D: 189,–	Interpreter	Disketten-Laufwerk, ggf. Vortex-Erweit.	ADL	k.A.	36 Seiten (e)
Modula 2	D: 499,–	Compiler	1 MByte-Laufwerk, Vortex-Erweiterung	ADL	Hochstrasser Computing	(e)
Pascal	D: 99,–	Compiler	Disketten-Laufwerk, ggf. Vortex-Erweit.	ADL	Ellis	(e)
Pascal	D: 215,–, K: 199,–	Compiler	Disketten-Laufwerk, keine	CPC	Hisoft	96 Seiten (d)
Pascal	D: 159,–	Compiler	Disketten-Laufwerk	ADL	Hisoft	80 Seiten (e)
Pascal 80	D: 159,90	Compiler	Disketten-Laufwerk	PRO	Hisoft	90 Seiten (e)
Small C	D: 148,–	Compiler	Disketten-Laufwerk 64 KByte-Erweiterung	MAR	Markt & Technik	200 Seiten (d)
Superpack 80	D: 141,90, K: 128,90	Assembler	Disketten-Laufwerk, keine	PRO	Profisoft	19 Seiten (d)
The Code Machine	K: 79,90	Assembler	keine	THO	Picturesque	68 Seiten (e)
Turbo-Pascal	D: 226,–	Compiler	Disketten-Laufwerk	MAR, HEI	Borland	(d)
Turtle Graphic	D: 49,–, K: 49,–	Interpreter	Disketten-Laufwerk, keine	GEP	GEPO Soft	15 Seiten (d)
Zen	K: 79,–	Assembler	keine	PRI	k.A.	(e)

Schneider CPC 664

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Basic	integriert	Interpreter	keine	CPC	Schneider	zirka 400 Seiten (d)
Basic-Compiler	D: 79,90	Compiler	keine	RUS	k.A.	(d)
C	D: 159,90	Compiler	keine	PRO	Hisoft	(e)
C	D: 169,–	Compiler	keine	ADL	Hisoft	120 Seiten (e)
C	D: 189,–	Compiler	ggf. Vortex-Erweit.	ADL	Software Toolworks	48 Seiten (e)
CBasic	D: 199,50	Compiler	keine	SDA	Digital Research	350 Seiten (e)
Cobol	D: 129,–	Compiler	keine	ADL	Ellis	165 Seiten (e)
Cobol	D: 189,–	Compiler	keine	SDA	Ellis	165 Seiten (e)
DEVPAC	D: 145,–, K: 129,–	Assembler	keine	CPC	Schneider	60 Seiten (d)
Dr. Logo	D: liegt bei	Interpreter	keine	CPC	Schneider	zirka 25 Seiten (d)
Forth	D: 189,–	Compiler	keine	SDA	Abersoft/Amstrad	60 Seiten (e)
Fortran	D: 129,–	Compiler	ggf. Vortex-Erweit.	ADL	Ellis	214 Seiten (e)
Fortran	D: 189,–	Compiler	keine	SDA	Ellis	214 Seiten (e)
Lisp	D: 189,–	Interpreter	ggf. Vortex-Erweit.	ADL	k.A.	36 Seiten (e)
Modula 2	D: 499,–	Compiler	1 MByte-Laufwerk, Vortex-Erweiterung	ADL	Hochstrasser Computing	(e)
Pascal	D: 99,–	Compiler	ggf. Vortex-Erweit.	ADL	Ellis	(e)
Pascal	D: 215,–, K: 199,–	Compiler	keine	CPC	Hisoft	96 Seiten (d)
Pascal	D: 159,–	Compiler	keine	ADL	Hisoft	80 Seiten (e)
Pascal 80	D: 159,90	Compiler	keine	PRO	Hisoft	90 Seiten (e)
Superpack 80	D: 141,90	Assembler	keine	PRO	Profisoft	19 Seiten (d)
Turbo Pascal	D: 226,–	Compiler	keine	MAR, HEI	Borland	(d)
Turtle Graphic	D: 49,–	Interpreter	keine	GEP	GEPO Soft	15 Seiten (d)

Schneider CPC 6128

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Basic	integriert	Interpreter	keine	CPC	Schneider	zirka 400 Seiten (d)
Basic-Compiler	D: 79,90	Compiler	keine	RUS	k.A.	(d)
C	D: 159,90	Compiler	keine	PRO	Hisoft	(e)
C	D: 169,–	Compiler	keine	ADL	Hisoft	120 Seiten (e)
C	D: 189,–	Compiler	ggf. Vortex-Erweit.	ADL	Software Toolworks	48 Seiten (e)
C-Compiler	D: 199,–	Compiler	Disketten-Laufwerk mit 128, RAM 64 KB.	GEP	GEPO Soft	200 Seiten (d)
CBasic	D: 199,50	Compiler	keine	SDA	Digital Research	350 Seiten (e)
Cobol	D: 129,–	Compiler	keine	ADL	Ellis	165 Seiten (e)
Cobol	D: 189,–	Compiler	keine	SDA	Ellis	165 Seiten (e)
CP/M Pascal-Compiler	D: 158,90	Compiler	keine	DRE	Hisoft	zirka 100 Seiten (e)
CP/M-C-Compiler	D: 158,90	Compiler	keine	DRE	Hisoft	100 Seiten (e)
Dr. Logo	D: liegt bei	Interpreter	keine	CPC	Schneider	zirka 25 Seiten (d)
DEVPAC	D: 145,–, K: 129,–	Assembler	keine	CPC	Schneider	60 Seiten (d)

Forth	D: 189,–	Compiler	keine	SDA	Abersoft/Amstrad	60 Seiten (e)
Fortran	D: 129,–	Compiler	ggf. Vortex-Erweit.	ADL	Ellis	214 Seiten (e)
Fortran	D: 189,–	Compiler	keine	SDA	Ellis	214 Seiten (e)
Lisp	D: 189,–	Interpreter	ggf. Vortex-Erweit.	ADL	–	36 Seiten (e)
Oxford Pascal	D: 149,–	Compiler	keine	RUS	Oxford Computer Systems	(d)
Pascal	D: 99,–	Compiler	ggf. Vortex-Erweit.	ADL	Ellis	(e)
Pascal	D: 215,–, K: 199,–	Compiler	keine	CPC	Hisoft	96 Seiten (d)
Pascal	D: 159,–	Compiler	keine	ADL	Hisoft	80 Seiten (e)
Pascal 80	D: 159,90	Compiler	keine	PRO	Hisoft	90 Seiten (e)
Pascal MT+	D: 199,50	Compiler	keine	SDA	Digital Research	270 Seiten (e)
Small C	D: 148,–	Compiler	keine	MAR	Markt & Technik	200 Seiten (d)
Superpack 80	D: 141,90	Assembler	keine	PRO	Profisoft	19 Seiten (d)
Turbo Pascal	D: 226,–	Compiler	keine	MAR, HEI	Borland	(d)
Turtle Graphic	D: 49,–	Interpreter	keine	GEP	GEPO Soft	15 Seiten (d)

Joyce

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
CBasic	D: 199,50	Compiler	keine	SDA	Digital Research	350 Seiten (e)
Cobol	D: 189,–	Compiler	keine	SDA	Ellis	165 Seiten (e)
Dr. Logo	D: liegt bei	Interpreter	keine	CPC	Schneider	zirka 25 Seiten (d)
Mallard-Basic	D: liegt bei	Interpreter	keine	CPC	Schneider	zirka 400 Seiten (d)
Pascal MT+	D: 199,50	Compiler	keine	SDA	Digital Research	270 Seiten (e)
Turbo Pascal	D: 226,–	Compiler	keine	MAR, HEI	Borland	(d)

Spectrum

Programmname	Datenträger/ Preis (Mark)	Art der Sprache	Hardware-Voraussetzungen	Bezugsquelle	Hersteller	Handbuch-Umfang
Aspect	K: 27,90	Assembler	keine	THO	Bug-Byte	9 Seiten (e)
Blast	K: 98,90	Compiler	k.A.	THO	Oxford Computer Systems	31 Seiten (e)
C	K: 98,– bis 100,–	Compiler	48 KByte	ACC, DRE, PRO, THO	Hisoft	78 Seiten (e)
Editor/Assembler	K: 32,– bis 34,–	Assembler	keine	PRO, PRI	Profisoft	34 Seiten (d)
Fith	K: 39,90	Interpreter	keine	PRI	k.A.	(e)
Forth	K: 57,– bis 59,–	Compiler	48 KByte	DRE, PRI	Sinclair	60 Seiten (e)
Forth	K: 49,–	Compiler	keine	ACC	Artic Computing	48 Seiten (e)
Forth	K: 59,90	Compiler	k.A.	THO	Artic Computing	48 Seiten (e)
FP Basic-Compiler	K: 75,90	Compiler	keine	DRE	Individual Software	4 Seiten (e)
M-Coder	K: 33,–	Assembler	k.A.	PRI	k.A.	(e)
M-Coder II	K: 39,90	Compiler	k.A.	THO	P.S.S.	7 Seiten (e)
Pascal 4T	K: 79,–	Compiler	48 KByte	ACC	Hisoft	98 Seiten (d)
Pascal	K: 74,90	Compiler	48 KByte	DRE	Hisoft	zirka 60 Seiten (e)
Pascal	K: 99,–	Compiler	48 KByte	PRI, RUS	Hisoft	zirka 60 Seiten (e)
Pascal	K: 99,90	Compiler	k.A.	THO	Hisoft	79 Seiten (e)
SPDE	K: 23,90	Disassembler	k.A.	THO	Campbell Systems	1 Seite (e)
The Colt	K: 49,90	Compiler	k.A.	THO	Hisoft	33 Seiten (e)
White Lightning (Forth-Compiler)	K: 59,90 bis 63,90	Compreter	48 KByte	PRO, THO	Oasis Software	132 Seiten (e)

Bezugsquellen

ACC Computer Accessoires, Jägerweg 10, 8012 Ottobrunn
ADL Adler-Computertechnik, Elisabethstr. 5a, 5800 Hagen 1
APP Apple, Ingolstädter Str. 20, 8000 München 45
ATA Atari Corporation, Frankfurter Str. 89 – 91, 6096 Raunheim
BRA Brainware, Kirchgasse 24, 6200 Wiesbaden
CPC Schneider Computer Division, Silvastr. 1, 8939 Türkheim
COM Compy-Shop, Gneisenaustr. 29, 4330 Mülheim/Ruhr
DAB Data Becker, Merowingerstr. 30, 4000 Düsseldorf
DRE H.G. Dreeser, Im Rosenhag 6, 5300 Bonn 1
GEP GEPO Soft, Gertrudenstr. 31, 4220 Dinslaken
HEI Heimsoeth Software, Frankfurterstr. 13, 8000 München 5
HIB HIB-Computerladen, Postfach 21 01 25, 8500 Nürnberg 21
HIL Philgerma, Ungererstr. 42, 8000 München 40
HOF Hofacker Verlag, Tegernseer Str. 18, 8150 Holzkirchen
JOL Jöllenbeck, Im Dorf 5, 2730 Weertzen
MAR Markt & Technik, Hans-Pinsel-Str. 2, 8013 Haar
PAN Pandasoft, Uhlandstr. 195, 1000 Berlin 12
PHI Philips, Mönckebergstr. 7, 2000 Hamburg 1
PRI Printadress, Postfach 15 – 73, 3548 Arolsen
PRO Profisoft, Sutthauser Str. 50 – 52, 4500 Osnabrück
RUS Rushware, An der Gümpgesbrücke 24, 4044 Kaarst 2
SAY Sanyo Büroelectronic, Truderinger Str. 13, 8000 München 80
SDA Schneider Data, Rindermarkt 8, 8050 Freising
SON Sony Deutschland, Hugo-Eckener-Str. 20, 5000 Köln 30
THO Thomas Wagner, Postfach 11 22 43, 8900 Augsburg

Programmiersprachen unter CP/M 80 für den C 128 PC und Schneider

Produktname	Preis	Datenträger	Art der Sprache	Anbieter
Turbo-Pascal 3.0	225,72	Diskette	Compiler	Markt&Technik
C-Basic-Compiler	174,-	Diskette	Compiler	Markt&Technik
Small-C-Entwicklungssystem	148,-	Diskette	Compiler	Markt&Technik
Pascal/MT	174,-	Diskette	Compiler	Markt&Technik
Turbo-Lader-Grundpaket	138,-	Diskette	-	Markt&Technik
Turbo-Lader-Business	148,-	Diskette	-	Markt&Technik
Turbo-Lader-Science	189,-	Diskette	-	Markt&Technik
Turbo-Pascal-3.0 grafikunterstützt	285,-	Diskette	-	Markt&Technik
Turbo-Grafik	225,72	Diskette	-	Markt&Technik
Turbo-Toolbox	225,72	Diskette	-	Markt&Technik
Hisoft-C-Compiler	160,-	Diskette	Compiler	Markt&Technik

Programmiersprachen unter CP/M-80

Produktname	Preis	Datenträger	Art der Sprache	Anbieter
ADA-Compiler	1162,80	Diskette	Compiler 80	Markt&Technik
C Basic	1881,-	Diskette	P-Code Interpreter	Markt&Technik
C Basic	564,30	Diskette	Compiler	Markt&Technik
MS Basic	1615,38	Diskette	Interpreter	Markt&Technik
MS Basic	1429,56	Diskette	Compiler	Markt&Technik
BDS-C	564,30	Diskette	Compiler	Markt&Technik
Supersoft C-80	1356,60	Diskette	Compiler	Markt&Technik
Cobol Level II	3846,36	Diskette	Compiler	Markt&Technik
MS Cobol	2859,12	Diskette	Compiler	Markt&Technik
RM Cobol	2793,-	Diskette	Compiler	Markt&Technik
Forth 8080	855,-	Diskette	Interpreter	Markt&Technik
Forth Z80	855,-	Diskette	Interpreter	Markt&Technik
Fortran 80SS	1647,30	Diskette	Compiler	Markt&Technik
MS Fortran 80	2045,16	Diskette	Compiler	Markt&Technik
MS Multisp	818,52	Diskette	-	Markt&Technik
PL/I	1815,-	Diskette	Compiler	Markt&Technik



I. Lüke/P. Lüke
Turbo-Pascal
1985, 290 Seiten

Das vorliegende Buch ist eine nach didaktischen Gesichtspunkten aufgebaute Einführung in sämtliche Versionen (einschl. 3.0) auf allen verfügbaren Betriebssystemen (CP/M, CP/M-86, MS-DOS). Es bietet nicht nur eine Einführung in die Sprache, sondern auch in das reichhaltige Repertoire an Zusatzfunktionen (auch für Grafik, Farbe, Sound und Window-Technik sowie für direkten Speicherzugriff) und Zusatzbibliotheken (Turbo-Toolbox, Turbo-Lader).
Best.-Nr. MT 90150
ISBN 3-89090-150-6 **DM 49,-**



J. Purdum/T. Leslie/A. Stegemöller
Die C-Programmbibliothek
1. Quartal 1986, 361 Seiten

Dieses Buch erspart dem C-Programmierer Stunden mühseliger Kleinarbeit und hilft, effizientere Programme zu schreiben. Der erste Teil zeigt, wie man zu universellen Bibliotheksfunktionen kommt und gibt Tips, wie C noch wirkungsvoller eingesetzt werden kann. Der zweite Teil enthält eine Reihe ausführlich erklärter C-Funktionen als wertvolle Ergänzung Ihrer Programmbibliothek. Dazu gehören unter anderem ein Terminalinstallationsprogramm, mehrere Sortier-Algorithmen und ein Satz ISAM-Funktionen. Um die Anwendung der Funktionen zu verdeutlichen, enthält das Buch einige Programmbeispiele.
- Die Programmbibliothek wendet sich an Leser mit Grundkenntnissen von C. Die gezeigten Programme und Funktionen sind so gehalten, daß sie rechnerund compilerunabhängig eingesetzt werden können.

Best.-Nr. MT 90133
ISBN 3-89090-133-6 **DM 69,-**

Markt & Technik-Fachbücher erhalten Sie bei Ihrem Buchhändler.



Unternehmensbereich Buchverlag
Hans-Pinsel-Straße 2, 8013 Haar bei München

Depot-Händler

Tragen Sie Ihre Buchbestellung auf eine Postkarte ein und schicken diese an einen Depothändler in Ihrer Nähe oder an Ihren Buchhändler.

Buchhandlung Herder, Kurfürstendamm 69
1000 Berlin 15, Tel. (0 30) 8 83 60 02,
BTX *921782#
Computare Fachbuchhandlung, Keithstraße 18
1000 Berlin 30, Tel. (0 30) 2 13 90 21
Thalia Buchhaus, Große Bleichen 19
2000 Hamburg 36, Tel. (0 40) 3 00 60 60
Boysen + Maasch, Hermannstraße 31
2000 Hamburg 1, Tel. (0 40) 3 00 50 50
Electro-Data, Wilhelm-Heidsiek-Straße 1
2190 Cuxhaven, Tel. (0 47 21) 6 12 88
Buchhandlung Muehlau, Holtenauer Straße 116
2300 Kiel, Tel. (04 31) 8 50 86
ECL, Norderstraße 94-96
2390 Flensburg, Tel. (04 61) 2 81 81
Buchhandlung Weiland, Königstraße 79
2400 Lübeck, Tel. (04 51) 7 40 06-09
Buchhandlung Storm, Langenstraße 10
2800 Bremen 1, Tel. (04 21) 32 16 23
Buchhandlung Lohse-Eissing, Marktstraße 38
2940 Wilhelmshaven, Tel. (0 44 21) 4 16 87
Buchhandlung Schmorl u. v. Seefeld, Bahnhofstraße 13
3000 Hannover 1, Tel. (05 11) 32 76 51
Buchhandlung Graff, Neue Straße 23
3300 Braunschweig, Tel. (05 31) 4 92 71
Deuerlich'sche Buchhandlung, Weender Straße 33
3400 Göttingen, Tel. (05 51) 5 68 68
Buchhandlung an der Hochschule, Holländische Straße 22
3500 Kassel, Tel. (05 61) 8 38 07
Stern Verlag, Friedrichstraße 24-26
4000 Düsseldorf, Tel. (02 11) 37 30 33
Buchhandlung Baedeker, Kettwiger Straße 33-35
4300 Essen 1, Tel. (02 01) 22 13 81
Regensberg'sche Buchhandlung, Alter Steinweg 1
4400 Münster, Tel. (02 51) 4 05 41-5
Buchhandlung Acker, Johannisstraße 61
4500 Osnabrück, Tel. (05 41) 2 84 88
Buchhandlung Brockmeyer, Querenburger Höhe 281/Unicenter
4630 Bochum, Tel. (02 34) 70 13 60
Buchhandlung Meier + Weber, Warburger Straße 98
4790 Paderborn, Tel. (0 52 51) 6 31 72
Buchhandlung Phönix GmbH, Oberntorwall 25
4800 Bielefeld 1, Tel. (05 21) 5 83 06-38
Buchhandlung Gonski, Neumarkt 24
5000 Köln 1, Tel. (02 21) 21 05 28
Mayer'sche Buchhandlung, Ursulinerstraße 17-19
5100 Aachen, Tel. (02 41) 47 77-0
Buchhandlung Behrendt, Am Hof 5a
5300 Bonn 1, Tel. (02 28) 65 80 21
Buchhandlung Cusanus, Schloßstraße 12
5400 Koblenz, Tel. (02 61) 3 62 39
Akad. Buchhandlung Interbook, Fleischstraße 61-65
5500 Trier, Tel. (06 51) 4 35 96
Buchhandlung W. Finke, Kipdorf 32
5600 Wuppertal 1, Tel. (02 02) 45 42 20
Buchhandlung Balogh, Sandstraße 1
5900 Siegen, Tel. (02 71) 5 52 98-9
Buchhandlung Naacher, Steinweg 3
6000 Frankfurt 1, Tel. (0 69) 29 80 50
Buchhandlung Wellnitz, Lautenschlägerstraße 4
6100 Darmstadt, Tel. (0 61 51) 7 65 48
Buchhandlung Feller + Gecks, Friedrichstraße 31
6200 Wiesbaden, Tel. (0 61 21) 30 49 11
Ferber'sche UNI Buchhandlung, Seltersweg 83
6300 Gießen, Tel. (06 41) 1 20 01
Sozialwissenschaftliche Fachbuchhandlung, Friedrichstraße 24
6400 Fulda, Tel. (06 61) 7 50 77
Gutenberg Buchhandlung, Große Bleiche 29
6500 Mainz, Tel. (0 61 31) 3 70 11
Buchhandlung Bock + Seip, Futterstraße 2
6600 Saarbrücken, Tel. (06 81) 3 06 77
Buchhandlung Wilhelm Hofmann, Bismarckstraße 98
6700 Ludwigshafen, Tel. (06 21) 51 60 01
Buchhandlung Loeffler, B 1,5
6800 Mannheim 1, Tel. (06 21) 2 89 12
Buchhandlung Stahn, Bahnhofstraße 13
7000 Stuttgart 60, Tel. (07 11) 56 14 76
Buchhandlung am Markt, Kramstraße 6
7100 Heilbronn, Tel. (0 71 31) 6 86 82
UNI Buchhandlung Kellner + Moessner, Kaiserstraße 18
7500 Karlsruhe, Tel. (07 21) 69 14 36
Buchhandlung Roth, Hauptstraße 46
7600 Offenburg, Tel. (07 81) 2 20 87
Rombach Center, Bertholdstraße 10
7800 Freiburg, Tel. (07 61) 4 90 91
Fachbuchhandlung Hofmann, Hirschstraße 4
7900 Ulm, Tel. (07 31) 6 09 49
Schaufler Elektronik, Bachstraße 62
7980 Ravensburg, Tel. (07 51) 2 61 38
Buchhandlung Hugendubel, Marienplatz
8000 München 2, Tel. (0 89) 2 38 9-1
Computerbücher am Obelisk, Barerstraße 32-34
8000 München 2, Tel. (0 89) 28 23 83
Pele's Computerbücher, Schillerstraße 17
8000 München 2, Tel. (0 89) 55 52 29
Universitätsbuchhandlung Lachner, Theresienstraße 43
8000 München 2, Tel. (0 89) 52 13 40
Buchhandlung Schönhuber, Theresienstraße 6
8070 Ingolstadt, Tel. (08 41) 3 31 46/47
Computerstudio Gertrud Friedrich, Ludwigstraße 3
8220 Traunstein, Tel. (08 61) 1 47 67
Buchhandlung Pustet, Kl. Exerzierplatz 4
8390 Passau, Tel. (08 51) 5 69 45
Buchhandlung Pustet, Gesandtenstraße 6
8400 Regensburg, Tel. (09 41) 5 30 61
Buchhandlung Dr. Büttner, Adlerstraße 10-12
8500 Nürnberg, Tel. (09 11) 23 23 18
Computer-Center-Burger, Leimitzer Straße 11-13
8670 Hof, Tel. (0 92 81) 4 00 75
Sortiments- u. Bahnhofsbuchh. J. Strykowski, Bahnhofplatz 4
8700 Würzburg, Tel. (09 31) 6 43 89
Buchhandlung Pustet, Grottenau 4
8900 Augsburg, Tel. (08 21) 3 54 37
Kemptener Fachsortiment, Salzstraße 30
8960 Kempten, Tel. (08 31) 1 44 13

Belgien:
Eicher Micro & Personal Computer, Hünningen 56-58
B-4780 St. Vith, Tel. (0 80) 22 73 93

Luxemburg:
Librairie Promoculture, 14, rue Duchscher (Pl. de Paris)
L-1011 Luxembourg-Gare, Tel. 48 06 91, Telex 31 12

Schweiz:
Buchhandlung Meissner, Bahnhofstraße 41
5000 Aarau, Tel. (0 64) 24 71 61
Bücher Balmer, Neugasse 12
6300 Zug, Tel. (0 42) 21 41 41
Buchhandlung Enge, Bleicherweg 56
8002 Zürich, Tel. (01) 201 20 78
Buchhandlung Orell Füssli, Pelikanstraße 10
8022 Zürich, Tel. (01) 2 11 80 11
Freihofer AG, Wissenschaftliche Buchhandlung, Universitätsstr. 11
8033 Zürich, Tel. (01) 3 63 42 82
Buchhandlung am Rößlitor, Webergasse 5
9001 St. Gallen, Tel. (0 71) 22 87 28



Unternehmensbereich Buchverlag
Hans-Pinsel-Straße 2, 8013 Haar bei München

Impressum

Herausgeber: Carl-Franz von Quadt, Otmar Weber

Chefredakteur: Michael Scharfenberger (sc)
Stellv. Chefredakteur: Michael Lang (lg)
Redakteure: Christine Geißler (cg), Volker Everts (ev), Horst Brandl (hb), Andreas Hagedorn (hg); Petra Wängler, Eva Hierlmeier (Koordination)
Redaktionsassistenz: Monika Lewandowski (222)

Fotografie: Jens Jancke

Titelgestaltung: Heinz Rauner Grafik-Design

Layout: Leo Eder (Ltg.),
Sigrid Kowalewski (Cheflayouterin)
Rolf Raß, Katja Milles

Auslandsrepräsentation:
Schweiz: Markt & Technik Vertriebs AG, Kollerstrasse 3, CH-6300 Zug, Tel. (0 42) 41 56 56, Telex: 862329 mut ch
USA: M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; Tel. 415-366-3600, Telex 752-351

Manuskripteinsendungen: Manuskripte und Programmlistings werden gerne von der Redaktion angenommen. Sie müssen frei sein von Rechten Dritter. Sollten sie auch an anderer Stelle zur Veröffentlichung oder gewerblichen Nutzung angeboten worden sein, muß dies angegeben werden. Mit der Einsendung von Manuskripten und Listings gibt der Verfasser die Zustimmung zum Abdruck in von der Markt & Technik Verlags AG herausgegebenen Publikationen und zur Vervielfältigung der Programmlistings auf Datenträger. Mit der Einsendung von Bauanleitungen gibt der Einsender die Zustimmung zum Abdruck in von Markt & Technik Verlag AG verlegten Publikationen und dazu, daß Markt & Technik Verlag AG Geräte und Bauteile nach der Bauanleitung herstellen läßt und vertreibt oder durch Dritte vertreiben läßt. Honorare nach Vereinbarung. Für unverlangt eingesandte Manuskripte und Listings wird keine Haftung übernommen.

Produktionsleitung: Klaus Buck (180)

Anzeigenverkauf: Britta Fiebig (211)

Anzeigenverwaltung und Disposition:
Patricia Schiede (172)

Marketingleiter Vertrieb: Hans Hörl (114)

Vertriebsleitung: Helmut Grünfeldt (189)

Verlagsleiter M&T Buchverlag: Günther Frank (212)

Vertrieb Handelsauflage: Inland (Groß-, Einzel- und Bahnhofsbuchhandel) sowie Österreich und Schweiz: Pegasus Buch- und Zeitschriften-Vertriebs GmbH, Hauptstätter Str. 96, 7000 Stuttgart 1, Tel. (07 11) 64 83-0

Bezugsmöglichkeiten: Leser-Service: Telefon (089) 46 13-249. Bestellungen nimmt der Verlag oder jede Buchhandlung entgegen.

Bezugspreis: Das Einzelheft kostet DM 14,–.

Druck: SOV St.-Otto-Verlag GmbH, Laubanger 23, 8600 Bamberg

Verantwortlich: Für redaktionellen Teil: Michael Scharfenberger
Für Anzeigen: Ralph Peter Rauchfuß (126).

Vorstand: Carl-Franz von Quadt, Otmar Weber

Anschrift für Verlag, Redaktion, Vertrieb, Anzeigenverwaltung und alle Verantwortlichen:
Markt & Technik Verlag Aktiengesellschaft, Hans-Pinsel-Straße 2, 8013 Haar bei München, Telefon (089) 46 13-0, Telex 5-22 052

Telefon-Durchwahl im Verlag:

Wählen Sie direkt: Per Durchwahl erreichen Sie alle Abteilungen direkt. Sie wählen 0 89/46 13 und dann die Nummer, die in Klammern hinter dem jeweiligen Namen angegeben ist.

Aktionäre, die mehr als 25% des Kapitals halten: Otmar Weber, Ingenieur, München; Carl-Franz von Quadt, Betriebswirt, München; Aufsichtsrat: Dr. Robert Dissmann (Vorsitzender), Karl-Heinz Fanselow, Eduard Heilmayr



Auf Ihrem Weg zum professionellen Computer-Anwender werden Sie früher oder später ...

Computer persönlich

Das aktuelle Fachmagazin für Personal Computer.

■ Wenn Sie jetzt den Schritt vom Heim-Computer zur professionellen Anwendung eines Personal Computers planen ■ Wenn Sie beruflich oder privat bereits einen Personal Computer benutzen ■ Wenn Sie selbst professionell programmieren ■ Wenn Sie regelmäßig Informationen über das breite Produktangebot auf dem Personal Computer-Markt benötigen ■ Wenn Sie professionelle Hard- und Softwaretests suchen ■ Wenn Sie Ihr eigenes System möglichst effizient einsetzen wollen, dann ist »Computer persönlich« genau Ihre Zeitschrift.

Die konsequente Ausrichtung auf professionelle Anwendungen bietet Ihnen alle wichtigen Informationen.

Und das alle 14 Tage, mittwochs bei Ihrem Zeitschriftenhändler oder im Computer-Fachgeschäft. ■

PC Magazin

Die einzige Wochenzeitung ausschließlich für Personal Computer im IBM-Standard.

■ Wenn Sie an aktuellen und umfassenden Informationen über IBM-PCs und kompatible Systeme interessiert sind ■ Wenn Sie stets über die neuesten und effektivsten Anwendungen für den professionellen und privaten Bereich informiert sein wollen ■ Wenn Sie sich für Marktübersichten und ausführliche Testberichte über Hard- und Software interessieren ■ Wenn Sie sich mit CAD/CAM, Netzwerken und der Anbindung von PCs an Groß-EDV-Anlagen beschäftigen, dann ist »PC Magazin« genau auf Ihre Bedürfnisse zugeschnitten.

Die Spezialisierung auf IBM-PCs und Kompatible ermöglicht eine gezielte Berichterstattung und bietet genügend Raum, um auf Anwenderprobleme spezifisch eingehen zu können.

»PC Magazin« — jeden Mittwoch neu bei Ihrem Zeitschriftenhändler oder im Computer-Fachgeschäft. ■

... auf diese beiden Computer-Zeitschriften stoßen:

Zur Anforderung Ihres kostenlosen Probeexemplares einfach den nebenstehenden Gutschein ausfüllen, ausschneiden, auf eine Postkarte kleben oder in ein Kuvert stecken und einsenden an: **Markt & Technik, Verlag Aktiengesellschaft, Vertrieb, Postfach 1304, 8013 Haar bei München.**

GUTSCHEIN

für ein kostenloses Probeexemplar

Senden Sie mir die neueste Ausgabe der von mir angekreuzten Zeitschrift kostenlos als Probeexemplar.

☐ Wenn mir »Computer persönlich« zusagt und ich es regelmäßig weiterbeziehen möchte, brauche ich nichts zu tun: Ich erhalte »Computer persönlich« dann regelmäßig alle 14 Tage per Post frei Haus geliefert und bezahle pro Jahr nur DM 98,– statt DM 143,– Einzelverkaufspreis. Zustellung und Postgebühren übernimmt der Verlag. Dieses Angebot gilt nur in der Bundesrepublik Deutschland einschließlich West-Berlin. Das Abonnement verlängert sich nur dann zu den dann jeweils gültigen Bedingungen, wenn es nicht 2 Monate vor Ablauf schriftlich gekündigt wird.

☐ Wenn mir das »PC Magazin« zusagt und ich es regelmäßig weiterbeziehen möchte, brauche ich nichts zu tun: Ich erhalte mein »PC Magazin« dann regelmäßig jede Woche per Post frei Haus geliefert und bezahle pro Jahr nur DM 155,– statt DM 229,50 Einzelverkaufspreis. Zustellung und Postgebühren übernimmt der Verlag. Dieses Angebot gilt nur in der Bundesrepublik Deutschland einschließlich West-Berlin. Das Abonnement verlängert sich nur dann zu den dann jeweils gültigen Bedingungen, wenn es nicht 2 Monate vor Ablauf schriftlich gekündigt wird.

Name, Vorname

Straße

PLZ Ort

Datum 1. Unterschrift

Mir ist bekannt, daß ich diese Bestellung innerhalb von 8 Tagen bei der Bestelladresse widerrufen kann. Zur Wahrung der Frist genügt die rechtzeitige Absendung des Widerrufs. Ich bestätige dies durch meine 2. Unterschrift.

Datum 2. Unterschrift

HCS586

Neueste Software für den Commodore 128 PC:

PROTEXT

Die Profi-Textverarbeitung im 128er-Modus mit vollautomatischer Silbentrennung, integrierter Tabellenkalkulation und Zusatzprogramm zum Überprüfen der Rechtschreibung.

PROTEXT ist ein leicht bedienbares Textprogramm mit hoher Leistungsfähigkeit. Eingebaute Hilfefunktionen ermöglichen eine schnelle Einarbeitung. Mit **PROTEXT** sind daher auch Anfänger in der Lage, alle Vorteile eines professionellen Textprogramms zu nutzen.

Was **PROTEXT** alles kann:

- Farbkombination für Hintergrund und Schrift (Vordergrund) frei wählbar;
- formatierte Ausgabe auf Bildschirm und Drucker mit programmierbaren Haltepunkten über serielle, V24- oder zwei Software-Centronics-Schnittstellen;
- vielfältige Formatanweisungen: linker/rechter Rand, vollautomatische Silbentrennung, Kopf-/Fußzeilen, Fußnoten, Zentrieren usw.
- schnelle selbstlernende Textkorrektur mit deutschem (ca. 25000 Worte) Grundwortschatz sowie neun Kundenbibliotheken, die in Text umgewandelt, bearbeitet, ergänzt, sortiert und ausdruckbar sind;
- Textübertragung per DFÜ mit Space-Optimierung und automatischer Fehlerkorrektur;
- leistungsfähige Rechenmöglichkeiten mit Zeilenmarkierung (Rechentabulator), Kolonnenverarbeitung, programmierter Tabellenkalkulation und Taschenrechner.

Hardwareanforderung:

- C 128 oder C 128 D
- 80-Zeichen-Monitor
- Commodore-Drucker oder Drucker mit Centronics-Schnittstelle

Best.-Nr. MD 254A

Zum sensationellen Preis von DM 89.- (sFr. 79.-/öS 990.-*)*

* inkl. MwSt.
Unverbindliche Preisempfehlung

TOPASS – Der ASE-Macroassembler für den Commodore 128 PC mit integriertem Editor, Monitor und Linker.



Dieser 6502-Macroassembler setzt neue Maßstäbe. Seine Leistungsfähigkeit wird auch den verwöhnten Maschinenprogrammierer überzeugen:

- integrierter Editor, der schon bei der Eingabe des Quelltextes eine Syntaxüberprüfung vornimmt;
- integrierter Linker, mit dem quellgesteuertes Linken von relokatiblen Modulen möglich ist;
- assemblereigene schnelle und gleichzeitig sehr leistungsfähige Integerarithmetik;
- über 2000 Labels können gleichzeitig verwaltet werden, das heißt Maschinenprogramme bis zu einer Länge von ca. 25 KByte Objektcode können bei Bedarf in einem Rutsch assembliert werden;
- Macros mit beliebig vielen Parametern, Macrobibliotheken, Minimacs, bedingte Assemblierung, Labeleingabe im Dialog, Ausgabe formatierter Assemblerlistings, Ausgabe sortierter Symboltabellen und vieles andere mehr.

Außerdem wird der ASE-Macroassembler von einem sehr guten Monitor und einem Relativlader unterstützt, der relokatible Module an beliebige Speicheradressen laden kann und endlich Schluß macht mit den Dutzenden Maschinenprogrammen auf Diskette, die sich nur durch ihre Startadresse unterscheiden!

Lernen Sie es kennen, das TOPASS Assembler-Entwicklungssystem! Es lohnt sich!

Best.-Nr. MD 253A

Für nur DM 89.- (sFr. 79.-/öS 990.-*)*

* inkl. MwSt.
Unverbindliche Preisempfehlung

Diese Markt&Technik-Softwareprodukte erhalten Sie in den Fachabteilungen der Kaufhäuser und in Computershops.

Wenn Sie direkt beim Markt&Technik Verlag bestellen wollen:
Nur gegen Vorauskasse, Verrechnungsscheck oder mit der eingedruckten Zahlkarte.



Unternehmensbereich Buchverlag
Hans-Pinsel-Straße 2, 8013 Haar bei München

Bestellungen im Ausland bitte an untenstehende Adressen:

Schweiz: Markt&Technik Vertriebs AG, Kollerstr. 3, CH-6300 Zug, Tel. 042/415656

Österreich: Ueberreuter Media Handels- und Verlagsges. mbH, Alser Str. 24, A-1091 Wien, Tel. 0222/481538-0